Mechanically Supported Design of Self-stabilizing Algorithms

Mechanisch Ondersteund Ontwerp van Zelf-stabiliserende Algoritmen
(met een samenvatting in het Nederlands)

Proefschrift
ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de Rector Magnificus, Prof. dr. J.A. van Ginkel ingevolge het besluit van het College van Decanen in het openbaar te verdedigen op maandag 9 october 1995 des octends te 10.30 uur.

door
Ignatius Sri Wishnu Brata Prasetya
geboren op 8 december 1966, te Jakarta

Promotor: Prof. Dr. S.D. Swierstra
Faculteit Wiskunde en Informatica

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Prasetya, Ignatius Sri Wishnu Brata

Mechanically Supported Design of Self-stabilizing Algorithms
Ignatius Sri Wishnu Brata Prasetya. - Utrecht:
Universiteit Utrecht, Faculteit Wiskunde en Informatica
Proefschrift Universiteit Utrecht. - Met index lit. opg.
- Met samenvatting in het Nederlands.
ISBN 90–393–1061–0
Trefw.: algoritmen / verificatie / wiskundige logica.

This thesis is dedicated to my mother, Mardiati,
Who never ceased to fight for what she believed in.
She is the most courageous woman I know,
The one person that I will never forget as long as I live.
I wish she were here to hold this book, and read this page.

I would also like to remember my father.
Abandoned by his own people,
He remains a brave soldier, even when all is lost.
–W.P.–

Preface

FOUR years ago, I was offered a research project. The theme was computer aided verification of programs —more specifically, distributed programs. I thought it was silly. From the software projects I did during my under-graduate time, I knew how hard it is to use a formal method to argue about the correctness of sizeable programs, and I could understand why industry considers formally proven correctness as impractical. Using a computer to check our proofs will not make it any easier because basically we still have to do all the thinking and supply the computer with all the details. Although there are tools to automatically generate proofs, such tools have their limitations. Anyway, I was intrigued by the project so I accepted the offer.

I was introduced to the theorem prover —or rather, theorem proving environment— HOL, a general purpose and interactive program in which we can add definitions and write proofs, which the program will verify. HOL has some tools to automatically construct proofs for certain formulas but, depending on the problem at hand, most of the time it was necessary to construct the proofs myself. Learning HOL was difficult. It is a machine with which we want to be able to handle formulas with all the delicacy of a human hand. To that end, HOL is equipped with lots of handles and buttons and we need to know them in detail to be able to operate HOL conveniently. Manipulating formulas through pulling handles and pushing buttons was certainly a quite different experience from how we are taught in school.

During the research I began to realize that perhaps we do not have much choice after all. We have become more and more dependent on computers. Accidents are bound to happen. If they happen with air planes, why should they not happen with software? Sooner or later, people will realize that they should impose a stronger standard on the quality of software. Employing theorem provers seems to be an idea worth investigating.

I discovered that verifying a distributed algorithm with a theorem prover is not as it seems at first sight. First, before we can even start verifying a program, we will have to supply the theorem prover with a wealth of mathematical facts which we are going to use. If a fact is not in the theorem prover's library of knowledge, we will have to prove it ourselves. This is a lot of work, and being a piece of new technology theorem provers generally, if not all, have a poorly developed library of facts. On the other hand, many distributed programs involve concepts from various branches of mathematics. Verifying them may require references to lots of other mathematical facts. The second problem is notation. With a few exceptions, most theorem provers only support an ASCII based format —or a slight variation thereof—, which is quite cumbersome when working with long and complicated formulas. Third, we leave many things implicit in our reasoning with pencil and paper, forced to concentrate on really important things. When working at the mechanical level of theorem provers, we need to make those implicit things in our reasoning explicit, discovering that we are often not aware of the extend of the implicitness in our reasoning.

There is nothing we can do to alleviate this third problem except learning through experience. The first two problems are however temporary. Things will get better as the technology evolves. I truly believe that there is a large potential use for theorem provers. We should be patient though, trying to systematically build their libraries of knowledge, to standardize them, and to provide good documentations. It will require highly qualified specialists to operate theorem provers, but I believe that in the end this is a small price to pay if we indeed plan to computerize the whole world, so to say.

In this thesis I would like to share with the reader, how designing a program and mechanically verifying it can be done hand in hand. My experiment with HOL quickly pointed out that the programming logic I was using, UNITY, was inadequate. So, I had to upgrade the logic and to verify it with HOL. A large portion of this thesis is concerned with this extension. In the remaining part of the thesis, a non-trivial example will be presented, and I will discuss my experience in verifying the example.

I would like to thank Doaitse Swierstra, my promotor (thesis supervisor), for offering me an AIO position, for giving me the precious chance to investigate the width and length of a theorem prover, for counselling me throughout my research, and for diligently proofreading this thesis and helping me writing the Samenvatting (summary) of this thesis. Although not directly related to my research, I also want to thank him for showing me the merits and potential of functional programming, especially in education. He gave me ideas which I hope I will have a chance to realize. I would like to thank the Woensdag Middag Club, especially for showing me the elegance of the Relational Calculus. I wish I had more time to learn more about it, and I certainly look forward to such an opportunity in the future. I would like to thank Rob Udink, with whom I have had many discussions regarding UNITY, and from whom I have learned much. I would like to thank the HOL community for their help and unspoken encouragement during the first two years of my research, during which I was but a helpless novice in HOL. I would like to thank the whole Vakgroep Informatica for accepting me —a strange outlander from the East I must have been— among them. I would also like to thank the reviewers of my thesis, in alphabetical order: Prof. Dr. E. Brinksma, Dr. F. de Boer, Prof. Dr. J. van Leeuwen, Prof. Dr. J.J.Ch. Meyer, and Prof. Dr. M. Rem, for taking the time and effort to review this thesis. I also thank Tanja Vos for proofreading this thesis. I thank Peter, Ina, Tim, Pieter, and Annelies for being my family here and providing me a safe haven in this land of tall people.

Finally, I would like to thank Ramosta for her continual support, and for adding alien colors to my life —strange though they are, but a pleasant change for someone whose world is dominated by a computer screen. I thank the God for His blessing —the row of luck granted to me could not be, so to say, the result of a random generator.

August 1995,

Wishnu Prasetya

The cover of this book depicts a dragon cruising a sky dominated by a sun. I like the picture, especially since there are many ways to interpret it. Here is one: the sun symbolizes hope. The dragon symbolizes mankind, travelling the sky, guided by the sun, seeking to discover what tomorrow hides from us.

Contents

The light in the eastern skies
Is still and always morning,
It alters the renewing air
Into belief and yearning.
—from The Lark, the Raven, and the Owl

Chapter 1 Introduction

THIS thesis presents the results of four years of research in which we explored the boundary of computer aided verification applied to the verification of distributed programs —more specifically, of the so-called self-stabilizing, distributed programs. It introduces an extension of the programming logic UNITY [CM88] as a mathematical framework for reasoning about such programs. It will be shown that the extension is compositional with respect to parallel composition[1]. The thesis then continues to describe how, for the purpose of computer aided verification, the extension may be embedded into an interactive theorem prover. As an experiment, we have also mechanically verified an example program. The example is a self-stabilizing and distributed algorithm for computing the minimal distance between all pairs of nodes in a network. We will discuss this example, and it will take a quite large portion of this thesis for several reasons. First, the problem is not as trivial as it it may seem. Second, to make the problem more interesting, and the results more useful, we will consider a generalized notion of minimal distance. Such a generalized notion of minimal distance is useful if, for example, we want to lift the algorithm so that it will also work for a hierarchically divided network. In such a network, nodes are grouped to form domains, and domains are grouped to form larger domains. The domains form a network of their own and also form a hierarchy. In fact, many large real networks are organized like that. Verifying the example turns out to require theories from various branches of mathematics, which is quite typical for many distributed algorithms. The computer aided verification of such a program will therefore have a quite different emphasis than, for example, a verification of an electronic circuit which typically involves a single, uniform kind of mathematics.

The theorem prover that we use to do the computer aided verification is HOL [GM93], which provides reasoning in higher order logic and programmable proof-tools.

The role of distributed programs has become increasingly important as more and more people hook their computers together, either locally or world-wide. The technology of computer networks advances rapidly and so is its availability. Today, it is no longer a luxury for a student to be able to quickly contact a fellow student, or a professor, or his future employer across the ocean through world-wide computer networks. There are even plans in some countries to make computer networks generally

[1] Meaning that the specification of a program is decomposable into specifications of its parallel components.

available at the house-hold level. Underlying this machinery, there are distributed programs which have to ensure that every message sent reaches its destination and have to provide management for resources shared by various users on various computers. These are very complicated tasks. Sooner or later, if not already, as we depend more and more on computer networks, we will have to seriously address the question of trustworthiness of the underlying distributed programs.

In practice, the correctness of a program is justified by testing it with various inputs. For complicated programs however, it soon becomes impossible to exhaustively test them. In his paper in *Mathematical Logic and Programming Language* [Goo85], a computer scientist, D.I. Good, sadly said the following about the practice of software engineering:

> *So in current software engineering practice, predicting that a software system will run according to specification is based almost entirely on subjective, human judgement rather than on objective, scientific fact.*

People have long been aware of this problem. In the 70's, pioneered by scientists such as E.W. Dijkstra, C.A.R. Hoare, and D. Gries, people began to advocate formal approach to the development of programs [Hoa69, Dij76, Gri81]. A program and its proof are to be developed hand in hand, and hence no post-verification is needed! However, the technique requires sophisticated mathematical dexterity not mastered by most computer engineers and students, not even today.

Just like programs, proofs are also prone to human errors. This is especially true for distributed programs. There is, we believe, no escape from this situation: distributed programs are inherently complicated and this fact, one way or another, will be reflected in their proofs. Refutation to proven 'correct' distributed programs occurs quite often. Even at the very abstract level mistakes can be made. An infamous example is perhaps the case of Substitution Axiom used in a programming logic called UNITY [CM88]. It was discovered that the axiom makes the logic unsound. A few years later a correction was proposed in [San91]. But even this turned out to be not entirely error-free [Pra94]. Although this kind of sloppiness occurs not too frequently, a mistake at the theory level may have severe consequences on the applications, especially if the faulty features are exploited to the extreme. Therefore, the need for computer aided verification is real.

Parallel to the formal approach to program derivation, technology to support computers aided verification was also developed. Roughly speaking, the technology can be divided into two mainstreams. One is called *model checking* or *simulation* and the other *interactive theorem proving*. In model checking [CES86] we have a computer which exhaustively traverses all possible executions of a program, extensively enough to be able to answer a given question about the program's behavior. The advantage is that we are relieved from the pain of constructing proofs. The technique works only for programs with a finite state-space and even then, it may not be feasible for a program with too large a state-space (the current technology is capable to deal with 10^{20} states [BCM+90]). However, this limit is quickly approached and surpassed, especially when dealing with sophisticated, infinitely large data-types. So, some intellectual effort may be needed nonetheless to reduce the original problem into one with a more manageable state-space.

In interactive theorem proving, we have a computer to interactively verify a proof. We basically have to construct the proof ourselves although most modern interactive theorem provers such as HOL [GM93] are also equipped with several facilities for partly automating the proof construction process. An interactive theorem prover usually provides a flexible platform as its underlying logic is extensible, thus enabling us to incorporate into the theorem prover the branch of mathematics required for a given application area. Modern interactive theorem provers are also based on powerful logics, supported by reasonably good notational mechanisms, enabling us to express complex mathematical objects and concepts easily and naturally.

Despite some of its advantages, model checking lacks the expressive power present in interactive theorem provers. What seems like a good solution is to extend interactive theorem provers with various automatic tools such as model checkers[2] (so, we would be able to consider a problem at a higher abstraction level and then decompose it into automatically provable units). People are currently working on this kind of integration. Some pointers that we can give are: [KKS91, Bou92, Bus94].

If the reader is interested in model checkers, a good starting point may be [CES86] or [BCM$^+$90]. In this thesis we will focus on interactive theorem proving, applied to the kind of problems described some paragraphs earlier.

An interactive theorem prover is usually based on some deductive logic. The computer will only accept a theorem if a proof, constructed by composing the logic's deduction rules, is supplied. Rigor is mandatory as it is also the only way to ensure correctness. However, this also means that we have to be very scrupulous in writing and manipulating formulas. Before we can even verify the simplest program with a theorem prover, we first needs to formalize and express the semantics of the programming language being used by giving a so called programming logic. That logic should of course be rich enough to express whatever aspect of programs we want to investigate. Once a choice has been made, basically all that we have to do is to embed the logic to the theorem prover. The embedding process itself is usually simple: it is a matter of translating the logic from its description on paper —let us call this 'human level description'— into the notation used by the theorem prover. The problem lies rather in the difference in the degree of rigor typically applied in a human level description and that required by a theorem prover. A human level description of a mathematical object or concept is typically intended to introduce some the object/concept to human readers. Some details may, intentionally or not, be omitted for a number of reasons, such as:

i. to improve the readability of formulas.

ii. the details are considered not interesting.

iii. the details can be extracted from the context.

iv. the details are considered as common knowledge.

[2] Note however, that the mandatory rigor imposed by a theorem prover would require that either the tools are first verified by the theorem prover or their results are 're-played' by the theorem prover.

v. the author is simply not aware of the details.

Using a theorem prover, on the other hand, requires that all details, interesting or not, are written down. If naively translated, the resulting logic may loose some strength. The deficiency may not be discovered immediately. When it is finally encountered we may have already produced thousands of lines of proofs, which may then need to be re-done. A great deal of effort is thus wasted. Being precise is the key, but this can be difficult, especially if we are so convinced that we know what we are talking about.

There are many interactive theorem provers. For our research we have chosen HOL, a system developed by M. Gordon [GM93]. HOL is based on Gordon's higher order logic[3]. There are other theorem provers with a more powerful logic. Nuprl [Con86] being an example thereof. Still, HOL's logic is certainly sufficient to deal with almost all kinds of applications. It is also extensible; that is, we can add new definitions or axioms. The main reason that we have chosen HOL is that it provides a whole range of proof-tools, which are also highly programmable. In addition, HOL is also one of the most widely-used theorem provers. Many users have made their work available for others, making HOL a theorem prover with, we believe, the greatest collection of standard mathematical theorems.

In this thesis we are especially interested in so-called self-stabilizing, distributed programs, and how to verify them in HOL. A self-stabilizing program[4] is a program which is capable to converge to some pre-defined equilibrium. Such a program is tolerant to perturbations made by the program's environment: if some perturbation throws the program from its equilibrium it simply re-converges to its equilibrium. Of course an essential assumption here is that following a period of unstability, the environment will eventually become stable, long enough for the self-stabilizing program to converge. The combined behavior of a self-stabilizing program and its environment is therefore usually specified relative to this period of stability.

There is a limitation which we must be aware of. In a self-stabilizing and distributed system, it is not possible to decide locally, for example in process a, whether the whole system has reached its equilibrium. Such a decision would require a to collect local information from other processes, which may be corrupted by the system's environment, resulting in a wrong conclusion being drawn by a. For example, a self-stabilizing program can only guarantee that a message sent by a will eventually reach b, but a will never know whether or not the message has arrived in b (although one can say that the probability increases with time). No acknowledgement mechanism will help. Therefore we will take a more general approach by considering programs that may be only partially self-stabilizing. A self-stabilizing program has an ability to revert to its equilibrium from any where in the state-space. A partially self-stabilizing program forbids some transitions as it might not be able to revert to its equilibrium after such a forbidden transition. This can, for example, be used to forbid the environment from corrupting acknowledgement information without being noticed.

[3] Roughly speaking, a higher order logic is a version of predicate calculus where it is allowed to quantify over functions

[4] The concept of *self-stabilization* was first conceived by E.W. Dijkstra [Dij74]

Self-stabilization has been applied to various problems, such as mutual exclusion protocols, communication protocols, and graph algorithms [BYC88, AG90, CYH91]. However, reasoning about self-stabilization is often complicated and it was not until recently that people attempted to deal with it more formally [Len93]. The need for formality is, first of all, to increase our confidence in our products, but also, as stated by the mathematician David Hilbert in 1900:

> *The very effort for rigor forces us to discover simpler methods of proof. It also frequently leads the way to methods which are more capable of development than the old methods of less rigor. —Quoted from [Gri90].*

In addition, we also have a very pragmatic need in mind: we need a formal approach to enable us to do verification using a theorem prover, and in an as easy way as possible.

Our starting point is UNITY, a programming logic developed by Chandy and Misra [CM88] for reasoning about distributed programs. Lentfert and Swierstra have extended the logic with a special operator to describe self-stabilizing properties of a program [LS93]. This thesis will start from there. Special attention will be given to the issues of inductive decomposition and compositionality. Both are useful in obtaining the local specification of a process from a global specification of a network of processes. Some examples will be presented to illustrate our style of calculation, especially concerning inductive decomposition. We believe style is important as it can make the difference between a concise and comprehensible proof and an either overly long or obscure one.

The developed logic has been embedded in the interactive theorem prover HOL mentioned some paragraphs earlier. It was then used to to mechanically verify the work in [Len93] on a self-stabilizing and distributed algorithm to compute the minimal distances between all pairs of vertices in a (hierarchical) network. To extend the applicability of the results, a generalized notion of 'minimal-distance' will be considered. Some price will have to be paid though. Various parameters will have to be introduced for the purpose of the generalization. Consequently, the size of formulas increases significantly, which may greatly obscure their meaning. A lot of algorithms actually have quite numerous parameters. It is just that in textbooks most of them, for the purpose of readability, are removed from formulas. When working with a theorem prover, rigor is mandatory and this informal kind of parameter hiding is not possible —at least not without the help of a special notational interface. Calculation, or verification, becomes more laborious although still within the range of practicality. But beyond that point, we may find the trustworthiness and the practicality of the method to be seriously in conflict. In our opinion, any modern, general purpose theorem prover should be supported by a decent notational interface. Nuprl is an outstanding example thereof. HOL still, unfortunately, lags behind in this respect. Further development of the technology of the notational interface will definitely be appreciated by everyone.

Although it is possible to present the complete verification of the program mentioned in the paragraph above, such would mean presenting thousands of lines of proofs and formulas. Instead, only some aspects will be exposed in this thesis. An outline of the verification will be given, and so are theorems which are considered important, or interesting.

When all is said and done, a frequently asked question about computer aided verification is: how much can we trust the computer we are using to verify the correctness of other computers? The answer is: not much actually. Attempts have been made to verify the programming languages used to implement the verification software. People are even trying to verify the compilers, and the chips used in the hardware. Still, we can never get an absolute guarantee that nothing can go wrong. A message sent by the proof-engineer's terminal to a remote file server may get scrambled unnoticed, for example. Or, some cyber criminal may alter the code unnoticed. In practical terms however, interactive theorem provers are very reliable, or at least, when faced with an extremely large and complicated verification task, it is our experience, over and over again, that they are much more reliable than men. That is however as far as such an extreme effort can be considered as an insurance.

1.1 Structure of this Thesis

This thesis will be divided into two parts. The first part is about the logic we are using. An elaborate motivation will be given in Chapter 2. Chapter 3 reviews some basic concepts of programming that are important for this thesis. These are, for example, the notions of write variables, predicates, and actions. Careful understanding of such basic concepts is important when one is about to embed a programming logic into some theorem prover. Chapter 4 presents an extension of the programming logic UNITY we mentioned earlier. UNITY has a problem that progress properties are not compositional with respect to parallel composition. That is, we cannot in general factorize a progress specification of a program into the specifications of its parallel components. This is too bad because we therefore cannot develop a component program in isolation. The extension presented in Chapter 4 is however compositional, and useful for the class of algorithms we are considering. Chapter 5 extends the logic further to deal with self-stabilization. The chapter also presents a number of laws to do inductive decomposition. Examples will be provided to illustrate our style of calculation. Emphasis will be placed on how various intuitive ideas relate to their formal counterparts. The results up to this point should be useful for designing self-stabilizing, distributed systems in general. In addition, almost all results have been mechanically verified, and available as HOL libraries.

One of the example, presented at the end of Chapter 5, is what we call Lentfert's Fair and Successive Approximation (FSA) algorithm [Len93]. The algorithm is a general algorithm to self-stabilizingly compute a certain class of functions called *round solvable* functions. Minimum-distance-like functions are examples of round solvable functions. It is the derivation of this algorithm which we took as a case study as we experimented with mechanical verification. It turned out that the hard part of the problem is, for non-trivial cases, not in verifying the algorithm itself, but rather in proving the round solvability of a function. In Chapter 6 special attention will be given to the round solvability of minimum-distance-like functions. Finally, in Chapter 7 the FSA algorithm will be lifted so that it will work for networks of 'domains' instead of ordinary networks of nodes. The domain-level FSA algorithm is a real example in which the general treatment to minimal distance functions presented in Chapter 6 will prove to be useful.

Although our objective is mechanical verification, Part I of this thesis should be of interest to those who are interested in formal techniques, applied to the design of distributed programs.

The second part of this thesis is about the embedding of the logic presented in Part I in the theorem prover HOL. Chapter 8 provides a brief introduction to the HOL system. Chapter 9 explains how the logic presented in Part I is embedded in HOL. Some examples illustrating how we represent programs and specifications in HOL, and how to do refinement in HOL, will be provided. Finally, we will show how the main results regarding the FSA algorithm and the round solvability of minimum-distance-like functions look like in HOL. An appendix will be included, providing a list of all relevant HOL definitions and most important (mechanically verified) HOL theorems produced during our research.

In Chapter 10 we present a conclusion we draw from our research.

1.2 A Note on the Presentation

Almost all theorems in this thesis are computer-checked. For such theorems there is basically little need to present their proofs. Still, the purpose of presenting a proof is not only so that it can be checked by its reader, but also:

i. to show how simple a theorem follows from some facts.

ii. to show a style of calculation.

iii. to give some insight in proving a closely related problem.

iv. to expose the actual nature of a theorem.

For these reasons, some proofs will be presented in this thesis. Should the reader still insist on checking the validity of the computer-checked results, the reader only needs to go through the list of definitions given in Appendix A. All of our computer-checked theorems are derived from these definitions. Their validity, with respect to these definitions, is guaranteed by HOL. However, it is indeed still possible that an absurd definition has been made.

When presented, computer checked results (definitions) will be marked by the names they are identified with in the HOL theories (definitions) that we wrote. For example:

Theorem 1.2.1 PINK PANTHER *Pink_Panther_thm*

$$Fu \;=\; Fu \circ Fu$$

The number and the name PINK PANTHER are how we refer to the theorem in this thesis. The name `Pink_Panther_thm` is how the theorem is called in HOL. *Implicitly,* *this means that the theorem is mechanically verified.* When referring to a theorem, definition, or equation we often —for the reader's convenience— include the page number in which the referred text can be found. The page number is printed as a subscript like in: Theorem $1.2.1_7$ or Theorem PINK PANTHER$_7$.

When presenting a theorem we often present it like this:

$$\frac{\begin{array}{c}A_1\\A_2\end{array}}{B}$$

which is another way for us to denote $A_1 \wedge A_2 \Rightarrow B$. The notation is commonly used within the UNITY community.

The reader will find a discrepancy between the notations used in the first and second part of this thesis. The purpose of Part I is to introduce to the reader the programming logic we use, and its various aspects. Comprehension takes higher precedence. The notation is therefore adapted to be as inspiring as possible. On many occasions, some parameters may be omitted from a formula and the reader should rely on the context when checking them. This should not be a problem.

In Part II, we describe how the logic is embedded in the theorem prover HOL. Definitions and theorems, as they are in the HOL notation, will be presented. The translation from the human notation from Part I to the HOL notation, and the converse, should be straight forward. However, we must warn the reader that the naming of variables and the order of parameters may be different as during the making of this thesis better naming and parameter ordering convention is also developed. Unfortunately, adapting our HOL code accordingly will take time we cannot afford at the moment.

Finally, should one want to apply the results presented in this thesis, the safe way to do it is *not* to consider them as presented in Part I, which after all was hand-typed and hence is likely to contain errors. Instead, one should take the results as they are reported by HOL. Some of them are listed in Appendix A. Admittedly they are in the ASCII format —which is of course nowhere close to, for example, LaTeXformatted formulas— but they are the output of HOL, as it is.

In the course of our research, thousands of lines of HOL proofs have been produced. A large part of the results should be reusable. There are for example libraries on well founded relations, program variables, transitive and disjunctive closures, lattices, and of course the extended UNITY logic that we used. All these libraries are available through ftp at `ftp.cs.ruu.nl`.

Part I

A Formal Approach to the Design of Self-Stabilizing Programs

Always before, you could explain
The turning darkness of the earth,
And how that dark embraced the rain,
And gave the ferns and flowers birth.
—from Kender Mourning Song

Chapter 2 Motivation

WE started our research with a pragmatic objective in mind: to mechanically verify the *Fair and Successive Approximation* (FSA[1]) algorithm, presented in a thesis by Lentfert [Len93]. The thesis was motivated by the *Smart Cabling Project* initiated by Utrecht University around 1988. The project aims to design an inherently safe packet switching network and serves as an experimental platform for formal program development methods. The algorithm is self-stabilizing —what this means will be explained later— and is actually a general algorithm. For example, Lentfert applied the algorithm for computing minimal distances between any pair of nodes in a (hierarchical) network. Lentfert's thesis contains a detailed, formal proof of the algorithm. Still, the proof is long and complicated so that it was felt that mechanical verification is needed. The interactive theorem prover HOL has been chosen for this purpose for the reasons explained in Chapter 1. In addition, once this goal is achieved, all theorems resulting from the verification would be in principle reusable.

Most modern interactive theorem provers such as HOL are equipped with tools to (partly) automate proofs[2]. However, we should not expect that everything can be verified automatically since the validity of HOL formulas has been known to be undecidable[3]. When automatic verification fails, we need to guide the theorem prover by hand. For example, the verification of the FSA algorithm requires a lot of hand-guided proofs[4].

During the course of our research, two things became obvious. First, the programming logic used by Lentfert misses certain low level details, and as a result it is actually not strong enough to derive some laws he was using, in particular laws concerning program composition. Second, because of the absolute rigor imposed by a theorem prover, a hand-guided, mechanical proof requires a great deal of effort, attention, and patience. The question of proof economy becomes therefore important. We learned that it is a good idea to lay out a complicated proof on paper before working on it with a theorem

[1] Not to be confused with 'Finite State Automata'.

[2] Examples of such automatic tools in HOL are decision procedures for Presburger natural number arithmetics, written by Boulton [Bou92], and for first order predicate calculus, written by Kumar, Kropf, and Schneider [KKS91].

[3] HOL, as mentioned in Chapter 1, is based on higher order logic. This logic contains first order logic, which, as shown by Church in 1936, is undecidable.

[4] Many distributed algorithms have higher order parameters which can severely hamper automation of their proofs.

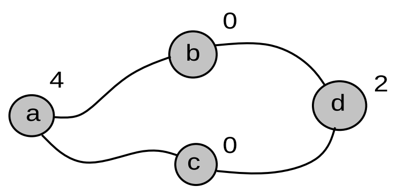


Figure 2.1: *A simple network.*

prover. Elegant proofs are definitely encouraged as this will reduce the effort required in verification.

Challenged by those two obstacles, we have worked on the logic used by Lentfert. We have added the necessary details to it, and also various calculational laws. Alongside with the (re-)invention of the laws, a style is developed, resulting in a shorter, more elegant proof of the FSA algorithm. Of course, we intend the logic to be generally applicable for designing and verifying distributed programs in general, and self-stabilizing, distributed programs in particular.

In the reminding of this chapter we will briefly discuss the aspects of the design and verification of self-stabilizing, distributed programs, which are relevant to our development of Lentfert's formalism. The discussion will be mostly informal as its purpose is only to provide a motivation for the rest of Part I. The motivation will be given with the aid of an example, which is a simplified version of Lentfert's algorithm.

2.1 Self-Stabilization

Imagine a network of nodes as in Figure 2.1. Ignore for the time being the numbers printed above the circles. The network in Figure 2.1 is *connected*, that is, each node is reachable from any other node. A non-empty sequence of nodes in such a network describes therefore a *path* from the first to the last node. Let us define the *distance* of a path to be the length of the path minus 1. A *minimal path* from a node a to another node b is defined as a path from a to b with the smallest distance. The *minimal distance* between nodes a and b is defined as the distance of a minimal path between a and b. For example, abd and acd are minimal paths from a to d in the network in Figure 2.1 and the minimal distance between a and d is 2.

In a computer network, a minimal path can be used to approximate the 'best' route to send messages from one computer to another. Many computer networks are therefore supported by a program that constructs and maintains minimal paths. A network can be described by a pair (V, N) where V is a set consisting of all nodes in the network and N is a function such that for any node a, $N.a$ is the set of all 'neighbors' of a.

Let us consider a self-stabilizing algorithm for computing the minimal distances. The algorithm is displayed in Figure 2.2 [5], and is in fact an instance of the Lentfert's

[5] The algorithm was originally due to Tajibnapsis [Taj77]. Tajibnapsis' algorithm is however not

```
program MinDist
init    true
begin
do forever
   for all $a \in V$ do
      for all $b \in V$ do
         if $b = a$ then $d.a.b := 0$ else $d.a.b := \min\{d.a.b' + 1 \mid b' \in N.b\}$
end
```

Figure 2.2: *The MinDist algorithm*

FSA algorithm. Although the algorithm only computes minimal distances, it can easily be extended to also compute minimal paths (this will be discussed in Chapter 6).

Notice that the program has an initial condition true, which means that it will work correctly no matter in which state it is started. Consequently, if during its execution an external agent interferes with it and tampers with the values of d, we can consider this as if the program is re-started in a new initial state. Since the program works correctly regardless its initial state, it will also do so in this new situation. In addition, once all $d.a.b$'s become equal to the actual minimal distances, the situation will be maintained forever (or until the next interference by the environment). Such properties are called *self-stabilizing* properties and are clearly very useful.

Let $_P\vdash\ p \rightsquigarrow q$ mean that if p holds somewhere during an execution of P then eventually q will hold and remain to hold forever. Let Dist.$a.b$ denote the actual minimal distance between a and b. We can use $\rightsquigarrow$ to express the self-stabilizing property of MinDist:

$$_{\mathsf{MinDist}}\vdash\ \mathsf{true} \rightsquigarrow (\forall a, b : a, b \in V : d.a.b = \mathsf{Dist}.a.b) \tag{2.1.1}$$

The specification states that the program MinDist must eventually establish $(\forall a, b : a, b \in V : d.a.b = \mathsf{Dist}.a.b)$. We will not be able to prove this directly. We can however break the total progress into smaller progress-steps. Induction is usually required to combine these steps into the specified progress property. This is not always easy. For example naively applying an induction to the values of $d.a.b$ does not work because these values can increase or decrease during execution[6]. Look again at the Figure 2.1. The number printed above a node i, $i \in \{a, b, c, d\}$, denotes the initial value of $d.a.i$. Note that the value of $d.a.a$ will decrease whereas the value of $d.a.b$ and $d.a.c$ will increase. Even an already correct value can temporarily be made incorrect. For example $d.a.d$ initially contains a correct value. However, if the process responsible for maintaining $d.a.d$ is executed first it will assign 1 to $d.a.d$, which is not the correct final value.

self-stabilizing.

[6] Had we restricted $d.a.b$'s to initially have the value of ∞, it would have been easier to prove (2.1.1).

Indeed, induction is an important technique. In fact, self-stabilizing programs often require complicated inductive proofs (for example as in [CYH91, Len93]). In Chapter 5 we will discuss some inductive decomposition principles which we have found useful. Examples will be given. Through the examples we want to illustrate how various ideas are translated in an as natural as possible way, from the informal to the formal level. Emphasis is also given to the calculation style. As has been remarked, the combination of elegance and clarity in proofs has a direct consequence on the amount of effort required to mechanically verify them.

2.2 Compositionality

Another topic we want to address is *compositionality*. Consider again the program MinDist. We can implement it as a distributed program consisting of processes MinDist.a for all $a \in V$ where each MinDist.a maintains the variables $d.a.b$ for all $b \in V$. Even the process MinDist.a can be implemented as a distributed program consisting of processes MinDist.$a.b$, for all $b \in V$ where each process MinDist.$a.b$ does:

```
do forever
    if $b = a$ then $d.a.b := 0$ else $d.a.b := \min\{d.a.b' + 1 | b' \in N.b\}$        (2.2.1)
```

It would be nice if we could decompose a global specification such as specification (2.1.1) into specifications of component programs. This would enable us to design each component in isolation. To be able to do this kind of decomposition we need laws of the form:

$$\frac{(P \text{ sat spec1}) \ \wedge \ (Q \text{ sat spec2})}{P \otimes Q \text{ sat } (\text{spec1} \oplus \text{spec2})} \tag{2.2.2}$$

where P and Q are programs, $\otimes$ is some kind of program composition, and spec1 and spec2 are specifications. Such properties are called *compositional*. It enables us to split the specification of $P \otimes Q$ into the specifications of P and Q. In particular, we are interested in the case where $\otimes$ is some form of parallel composition.

There is another advantage of having compositional properties. It may be the case that the separation of spec1$\oplus$spec2 into spec1 and spec2 using (2.2.2) yields expressions that are more complicated and are, basically, harder to prove. Still, we just have to verify spec1 against P alone, instead of against both P and Q. The same thing holds for spec2. For sufficiently large P and Q the application of (2.2.2) often reduces the amount of proof obligations significantly.

An important property of any program is its progress. However, not much is known about the compositionality (with respect to parallel composition) of progress properties. It has been recognized that much can be concluded by just looking at which components read or write to which variables. Still, the matter seems to be not well understood. More precision is required for better understanding. Some formal theory about variables, how mundane it may sound, needs to be developed.

Let us now consider the following example. Let ${}_P\!\vdash p \mapsto q$ mean that if p holds during an execution of the program P then eventually q will hold. So, $\mapsto$ describes progress.

Let a, b, and c be boolean variables. Suppose now ${}_P\!\vdash a \mapsto c$ holds. The property does not refer to b, so we may expect that if we put P in parallel with Q defined below then the progress will be preserved.

```
Q: do forever $b := \neg b$
```

However, even though the expression ${}_P\!\vdash a \mapsto c$ does not refer to b, it may happen that the progress actually depends on b, for example if P is the following program:

```
P: do forever
   begin
   if $a$ then $b :=$ true else skip;
   if $b$ then $c :=$ true else skip
   end
```

In this case, Q will destroy the progress $a \mapsto c$. Still, if we put P in parallel with Q' which does nothing to a and c and only writes to b under the condition, say, C then we can conclude that the composite program will have the property $a \wedge \neg C \mapsto c \vee C$ [7]. That is, if $a \wedge \neg C$ holds during any execution of P in parallel with Q', then Q' cannot modify b as long as $\neg C$ holds, and then eventually c will hold by the actions of P. However, it is possible that P itself may set C to true, which explains the second disjunct in $a \wedge \neg C \mapsto c \vee C$.

The examples suggests that useful results can be drawn by recording the set of variables upon which a progress property depends. In this thesis we will take a less radical approach. We observe that the only part of a program that is ever influenced by its own actions is its writable part. We will therefore split a progress specification in two parts. The first part, the actual progress part, describes progress made on the write variables (and those variables only) of a program. The second part, the so-called J-part, describes the state of the other variables (consequently, the J-part is 'stable' during any (isolated) execution of the program). Especially interesting results can be obtained for programs that are *write-disjoint*. In this case the J-part of a specification of P also acts as a specification for the other programs which P is composed with.

Two programs are said to be *write-disjoint* if their sets of write variables are disjoint. For example the programs MinDist.$a.b$ in (2.2.1) are pair-wise write-disjoint.

Let $J\ {}_P\!\vdash p \mapsto q$ mean: (1) $p \mapsto q$ is a progress property of P through its writable part, and (2) the predicate J describes the state of the variables not writable by P and in addition J also fully describes the dependency of $p \mapsto q$ on these variables. Let P and Q be two write-disjoint programs such that $J\ {}_P\!\vdash p \mapsto q$ holds. Since P and Q are write-disjoint, Q cannot write to P's write variables. Consequently, J also fully describes the dependency of $p \mapsto q$ on Q, and hence if Q cannot destroy J neither can it

[7] In fact, this is an instantiation of the Singh Law [Sin89].

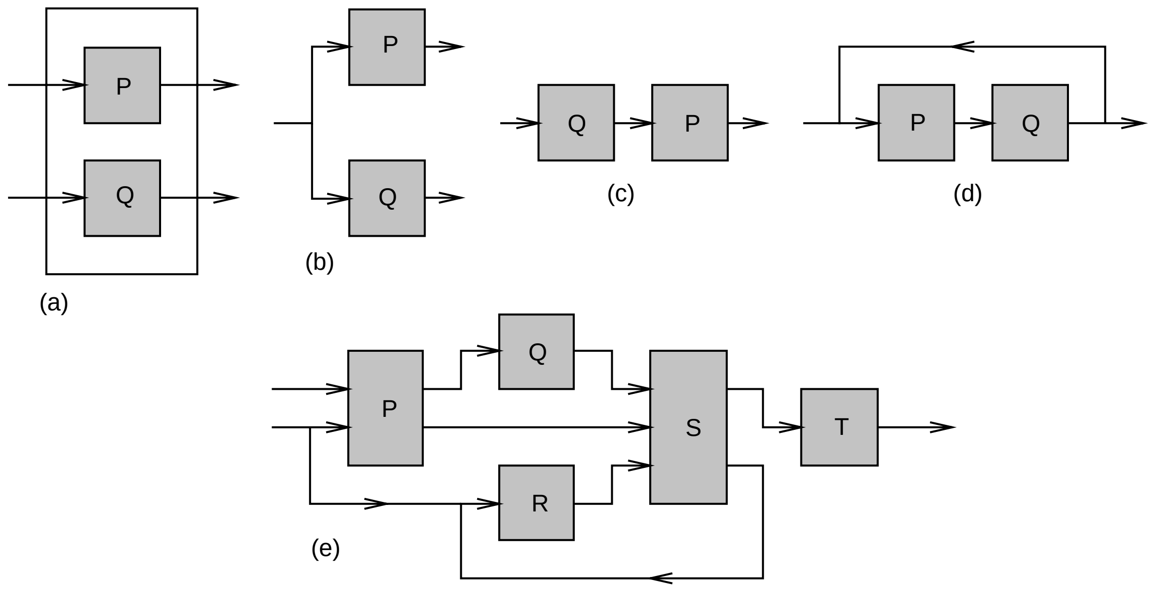


Figure 2.3: *Instances of the write-disjoint composition.*

destroy $p \mapsto q$. This property is called *transparency*. As we will see later, transparency turns out to be an important property in composing programs.

Let $\mathbf{w}P$ denote the set of all write variables of P. Let $P \div Q$ mean that P and Q are write-disjoint:

$$P \div Q \quad = \quad (\mathbf{w}P \cap \mathbf{w}Q = \emptyset) \tag{2.2.3}$$

Let $P [\!] Q$ denote the parallel composition of the programs P and Q. By this we simply mean that we put P and Q side by side and execute them concurrently. How the two programs communicate is not relevant at this point. If $P \div Q$ holds, then $P [\!] Q$ is called the *write-disjoint composition* of P and Q.

The transparency of progress properties mentioned earlier can be expressed by the following law:

$$\frac{(J \;_P\!\vdash p \mapsto q) \;\wedge\; \text{"Q cannot destroy J"}}{J \;_{P[\!]Q}\!\vdash p \mapsto q} \tag{2.2.4}$$

Note that the law has the form of (2.2.2). Using the law, we can assign the task of establishing a progress property, like $p \mapsto q$, to a write-disjoint component. A formal treatment of this kind of laws will be given in Section 4.8.

Composition of write-disjoint programs occurs frequently in practice (as in the program MinDist, for example). See Figure 2.3. Diagram (a) shows two programs in parallel, not interfering with each other. Diagram (b) shows two programs in a so-called *fork* construction, in which they share the same input variables, but have no common write variables. Diagram (c) shows two programs in a so-called *layering* construction. The program Q on the left is called *the lower layer*, which 'sends its output' to the *the upper layer P*. Layering —also called collateral composition— has been recognized by Herman [Her91] and Arora [Aro92] as an important technique for combining self-stabilizing programs. Diagram (d) shows how 'outputs' can be fed back in a write disjoint composition. Finally, diagram (e) shows a more complicated example of a network of write disjoint programs.

Communication through channels can also be modelled by write-disjoint composition. Consider the following programs sender and receiver:

```
prog   sender
read   $x, a!$
write  $x, a!$
init   $a! = []$
do forever
   begin
   $x :=$ "generate a new datum" ;
   $a! := x; a!$
   end
```

```
prog   receiver
read   $y, a!, a_{\#}$
write  $y, a_{\#}$
init   $(a! = []) \wedge (a_{\#} = 0)$
do forever
   begin
   if $a_{\#} < \ell.(a!)$
     then $y, a_{\#} := a![a_{\#}], a_{\#} + 1$ ;
   "process the new datum in y"
   end
```

The sender and receiver are connected by an asynchronous channel a. The channel is modelled by a history variable $a!$, recording all messages sent through a. The progress of the receiver is recorded by the variable $a_{\#}$ in such a way that the prefix $a![0 \ldots a_{\#})$ contains all the messages in $a!$ which have been received by receiver. Note that sender and receiver are write disjoint.

2.3 Hierarchical FSA Algorithm

We have mentioned that as a test case we want to mechanically verify Lentfert's FSA algorithm. An instance of the algorithm, shown in Figure 2.2, self-stabilizingly computes the minimal distance between any pair of nodes in an ordinary network.

We take up two challenges. First, we want to generalize the notion of 'minimal distance', and second, we want to consider the hierarchical version of the FSA algorithm. In this section we want to bring up some aspects of these challenges in order to give some insight to the reader about the nature of the problem.

The minimal distance between two nodes a and b (denoted by $\mathsf{Dist}.a.b$) in a connected network (V, N) can be characterized by the following equation:

$$(\mathsf{Dist}.a.a = 0) \;\wedge\; (a \neq b \Rightarrow (\mathsf{Dist}.a.b = \mathsf{min}\{\mathsf{Dist}.a.b' + 1 \mid b' \in N.b\})) \qquad (2.3.1)$$

The assumption here is that the 'distance' of going between two linked nodes a and b (that is, $b \in N.a$) is 1. This can be generalized by allowing the distance of two connected nodes (a, b) to be some natural number, called the *weight* of the link (a, b). This can be abstracted by a function $w : V \to V \to \mathbb{N}$ such that $w.a.b$ returns the weight of (a, b). This means that the function $+1$ in equation (2.3.1) is to be replaced by $+w.b'.b$.

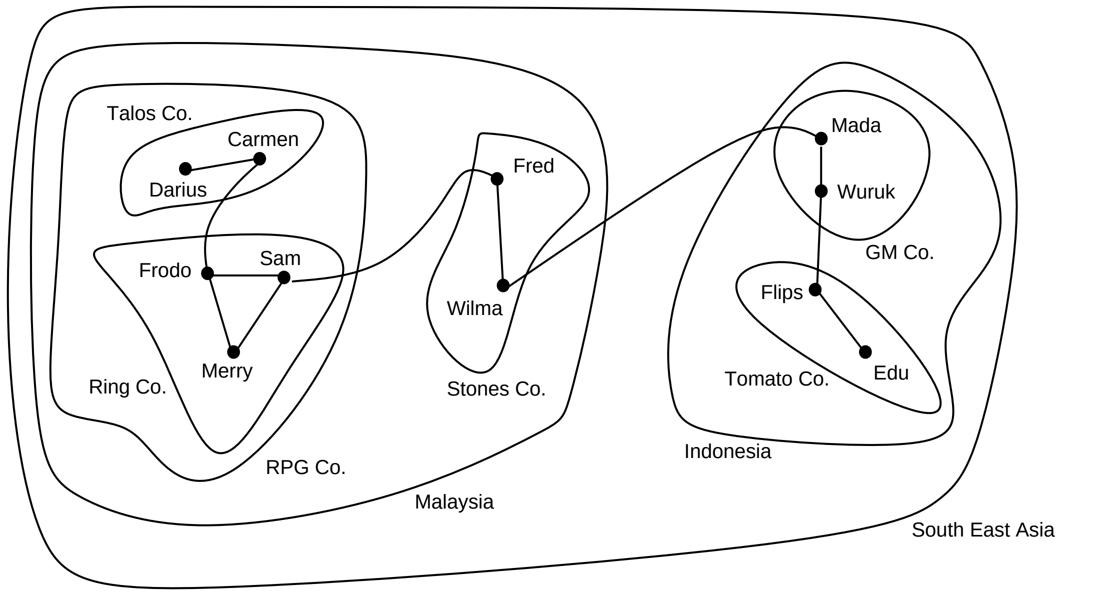


Figure 2.4: *South East Asia hierarchically divided network.*

Further generalization can be made by replacing $+w.b.b'$ with an arbitrary function $\mathsf{addW}.b'.b$, min by the greatest lower bound operator $\sqcap$ of some lattice, and 0 with the bottom element of the lattice[8]. So, equation (2.3.1) is now generalized to:

$$(\delta.a.a = \bot) \ \wedge \ (a \neq b \Rightarrow (\delta.a.b = \sqcap\{\mathsf{addW}.b'.b.(\delta.a.b') \mid b' \in N.b\})) \qquad (2.3.2)$$

Of course the choice of $\sqcap$ and addW cannot be arbitrary. We are interested in the conditions under which δ can be computed by Lentfert's FSA algorithm.

An application of (2.3.2) is to define the notion of hierarchical distance. First we have to explain what a hierarchical network is. An example of a *hierarchical network* is given in Figure 2.4. It displays an ordinary network with Darius, Sam, Flips, ... as nodes, connected to each other. The reader can think that these nodes represent real computers or computer users, connected by (physical) communication links. A hierarchical network is an ordinary network divided into *domains*. For example, Flips and Edu form the domain Tomato Co., and the domain Tomato Co. and GM Co. form a larger domain Indonesia. A domain represents a local network of computers. In addition, the domains form a tree-hierarchy, as shown in Figure 2.5. The domains have *levels*: the level of a root domain is 0, the level of a non-root domain A is the level of its father plus 1. Hierarchical division is very natural and occurs frequently in practice. Offices are hierarchically divided, governments are hierarchically divided, our inter-connected networks are hierarchically divided. Such a division is natural because it is our way to systematically group objects together.

[8] Alternatively, one can also generalize the problem using a regular algebra as in [BEvG94]

Hierarchical division is also introduced so that we can hide information. One may not want to know the exact geography of Indonesia, but may be satisfied with the knowledge that Indonesia lays in South East Asia and is a neighbor of Malaysia. If the network reflects an actual network of computers, a computer sending mail to Flips in Indonesia only needs to know how to send it to some node in Indonesia. It is then up the Indonesia's national network to deliver it to Flips. Indeed, if the network is very large, such as the world-wide Internet network, we may not want to store complete information about the whole network locally in each node. Instead, each node stores only information of *visible* domains. The exact meaning of 'visible' is unimportant for now, but the idea is: a domain may be visible, but not its interior. One of the consequences is that things have to be reasoned about at the level of domains instead of nodes.

In an ordinary network (V, N) a node b' is called a neighbor of b if $b' \in N.b$. This network at the node level reflects the physical network. It also defines a network at the domain level by defining a domain B' to be a neighbor of a domain B, denoted by $B' \in \mathcal{N}.B$, if there exists a node b' inside B' and a node b inside B such that $b' \in N.b$. Let say now we have some notion of visibility defined on the set of domains. Let $B' \curvearrowleft B$ denote B' *is visible from* B. The idea is to restrict $\mathcal{N}$ with this visibility relation. If we define $\mathcal{N}'.B = \mathcal{N}.B \cap \{B' \mid B' \curvearrowleft B\}$, the notion of minimal distance can be extended from the node level to the domain level by generalizing equation (2.3.2) to:

$$(\delta.A.A = \perp) \;\wedge\; (A \neq B \Rightarrow (\delta.A.B = \sqcap\{\mathsf{addW}.B'.B.(\delta.A.B') \mid B' \in \mathcal{N}'.B\})) \qquad (2.3.3)$$

Note that by definition of $\mathcal{N}'$, to compute $\delta.A.B$ we only need $\delta.A.B'$ of all neighboring (by $\mathcal{N}$) *and* visible domains B'!

In [Len93] it is proposed to define the minimum hierarchical distance such that the cost, or 'distance' of going from a domain B' to a neighboring domain B_1 is always higher than the cost of going to B_2 if the level of B_1 is lower than the level of B_2. This can be achieved by defining distances as vectors, and using the lexicographic ordering to define the $\sqcap$ used in the equation (2.3.3). A proper instantiation of addW can be chosen accordingly.

This would mean that results on the minimal distance algorithm for ordinary networks can be extended to apply to its hierarchical version. And of course, we also hope that the same things can be done to the FSA algorithm in general. Still, we have to take into account that a domain is not a single, non-divisible entity as a node is. It has sub-domains and nodes. A computation at the domain level will therefore have to be mapped to the nodes. All these aspects will be exposed in detail in Chapter 7.

2.4 Summary

We have explained what a hierarchical network is and motivated the practical interest for such a network. We have given an instance of the FSA algorithm, namely the minimal distance algorithm, we have explained what it does, and motivated how it

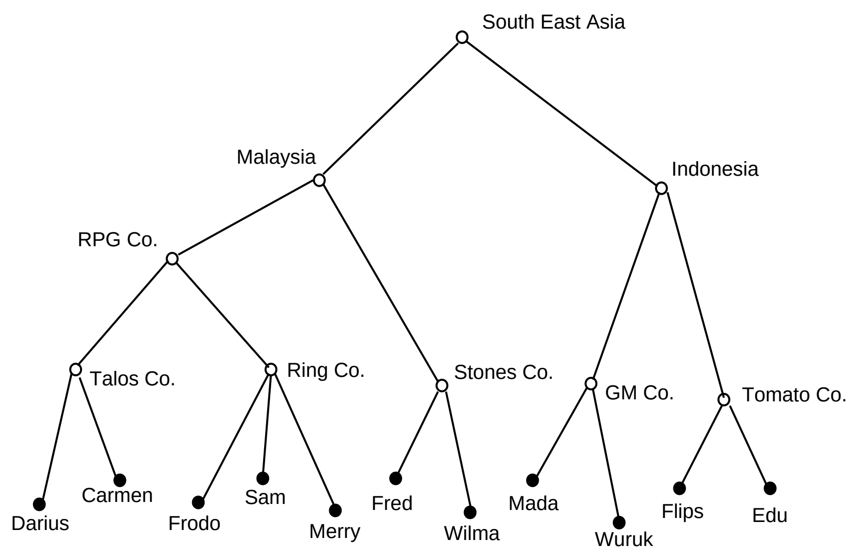


Figure 2.5: *The hierarchy of the domains within South East Asia.*

can be generalized to handle a hierarchical network. We have mentioned the self-stabilizing property of the minimal distance algorithm, and motivated the importance of compositionality. In the rest of Part I we are going to:

i. introduce a programming logic in which we can reason about self-stabilizing and distributed programs.

ii. introduce various calculation laws to facilitate the design and verification of distributed algorithms.

iii. investigate the compositionality of self-stabilization.

iv. present examples of inductive decompositions.

v. discuss the verification of Lentfert's FSA algorithm.

Past the point of no return,
No backward glances.
The games we played 'till now
Are at the end.
Past all thought of 'if' or 'when'.
No use resisting: a burden the thought,
And let the dream descend.
—the Phantom

Chapter 3 The Basics

THIS chapter explains the meaning of some most primitive concepts of programming. The concepts mostly deal with objects at the level of statements and lower. Most textbooks about formal methods are rather implicit in addressing these concepts, since they are thought to be well understood. However, in Chapter 4 we want to study conditions required to preserve progress properties under parallel composition. To thoroughly study the problem it is required that we are being explicit at the low level, especially with regard to various accessibility modes of variables, as motivated in Section 2.2. A minimal theory about accessibility needs to be developed. There is another reason for being explicit at the lowest level: it makes embedding the programming logic presented in Chapter 4 into the interactive theorem prover HOL straightforward.

We will explain the mathematical objects representing program-states, statements, and state-predicates. Some standard statements, operators on statements, a notion of statement-refinement, and Hoare triple specifications will be defined. It will also be defined what it means for a statement to have write and read access to a variable. In denotational semantics [Sch86] the meaning of a programming language is given in terms of interpretation functions that map constructs in the language to some interpretation domains. Here, we only present our interpretation domains[1]. Programming constructs will be explained in terms of these domains. No language will be explicitly given and we rely instead on 'commonly' accepted notations[2]. Confusion does sometimes arise. To help the reader, we will, throughout this thesis, explicitly write down the meaning of a potentially ambiguous expression.

[1] It has been said that our goal is to be able to mechanically verify distributed programs using the interactive theorem prover HOL. To this end the programming logic presented in Chapter 4 will be embedded in HOL. The logic that underlies HOL, its notational style, and the availability of basic libraries greatly influence the choice of the interpretation domains.

[2] In HOL, we usually encode programs directly in terms of the interpretation domains. A programming language can always be defined on top of the interpretation domains. In fact, we include a minimal language in our HOL package.

```
program Fizb
vars    $a, x, y$
do forever
    begin
    if $a = 0$ then $x := 1$ else skip ;
    if $a \neq 0$ then $x := 1$ else skip ;
    if $x \neq 0$ then $y, x := y + 1, 0$ else skip
    end
```

Figure 3.1: *The program* Fizb

3.1 States, Variables, and Values

A program has variables. Their values vary during the program's execution. The values of the program's variables at a given moment reflect the program's state at that moment. Therefore, program-states are usually represented by functions from variables to values. We will assume a universe Var of identifiers, large enough to represent *all* program variables, and a universe Val of *all* values these variables may take. So, any variable of any program P is a member of Var.

Let P be a program and V be a set consisting of all variables of P. A state of P can be represented by a function $s \in V \to \mathsf{Val}$. The value of a variable v in state s is given by $s.v$. However, since in HOL all functions are required to be total and since sub-typing is not possible, we will represent a state of P by a *total* function $s \in \mathsf{Var} \to \mathsf{Val}$. The value of a variable v outside V in a state s is irrelevant to P in the sense that it cannot influence any execution of P and neither can any execution of P influence it. The set of all program-states is denoted by State.

For example, consider the program Fizb in Figure 3.1. It has three variables a, x, and y. A state of Fizb is for example $s \in \mathsf{Var} \to \mathsf{Val}$ such that:

$$(s.a = 1) \wedge (s.x = 1) \wedge (s.y = 4) \wedge (\forall v : v \notin \{a, x, y\} : s.v = 0) \qquad (3.1.1)$$

The value of the variable a in the state s is $s.a$, which is equal to 1. The value of any variable outside $\{a, x, y\}$, hence the variable does not belong to Fizb, in the state s is 0, but this should not matter to Fizb.

For a function $f \in B \to C$, the projection or restriction of f to a set A, denoted by $f \upharpoonright A$ can be defined as a *partial* function of type $B \to C$ such that $(f \upharpoonright A).x = f.x$ if $x \in A \cap B$ and else it is undefined. However, we want the projection of a state to be a state again, which is a total function. For this purpose, a constant $\aleph$ is introduced. In HOL, all objects are typed. A type defines of course a set. The types in HOL are all non-empty. All that is known about $\aleph$ is that for any given (non-empty) type A, $\aleph_A$ exists and is a member of A. The constant $\aleph$ can be thought as representing 'undefinedness' in a partial function. Evaluating an 'undefined' value can be thought as an error and therefore special calculation rules have to be added to handle such errors.

We prefer to regard $\aleph$ as an ordinary value, representing the set of 'uninteresting' but otherwise valid values. This eliminates the need to introduce special rules to handle 'undefinedness'.

Here is our definition of projection and some of its useful properties:

Definition 3.1.1 PROJECTION *Pj_DEF*
For all $f \in A{\to}B$, $V \subseteq A$, and $x \in A$:

$$(x \in V \;\Rightarrow\; (f \upharpoonright V).x = f.x) \;\wedge\; (x \notin V \;\Rightarrow\; (f \upharpoonright V).x = \aleph)$$

Theorem 3.1.2 $\upharpoonright$ EXTENSION *Pj_EQ*

$$(f \upharpoonright V = g \upharpoonright V) \;=\; (\forall x : x \in V : f.x = g.x)$$

Theorem 3.1.3 $\upharpoonright$ ANTI-MONOTONICITY *Pj_EQ_MONO*

$$V \subseteq W \wedge (f \upharpoonright W = g \upharpoonright W) \;\Rightarrow\; (f \upharpoonright V = g \upharpoonright V)$$

Theorem 3.1.4 $\upharpoonright$ COMPOSITION

$$f \upharpoonright V \upharpoonright W \;=\; f \upharpoonright (V \cap W) \;=\; f \upharpoonright W \upharpoonright V$$

Theorem 3.1.5 $\upharpoonright$ EXTENSION BY $\cup$

$$(f \upharpoonright (V \cup W) = g \upharpoonright (V \cup W)) \;=\; (f \upharpoonright V = g \upharpoonright V) \wedge (f \upharpoonright W = g \upharpoonright W)$$

Corollary 3.1.6

$$(f \upharpoonright \emptyset = g \upharpoonright \emptyset) \;\wedge\; (f \in A{\to}B \Rightarrow (f \upharpoonright A = f))$$

3.2 Actions

A program consists of statements, also called transitions, or actions. From now on we will use the latter term. The execution of an action can bring the program to a new state. An action can be non-deterministic. That is, given a state, there are several possible new states which the action may result into. We model non-deterministic actions by relations on State with the following interpretation. Let $a \in \mathsf{State}{\to}\mathsf{State}{\to}\mathbb{B}$ be an action. If $a.s.t$ holds, it means that t is a possible new state, after executing a in the state s. If t is the only new state related to s, then the action a is deterministic at s and it *will* end in t if executed in s.

The set of all actions is denoted by Action. Here are some examples. The relation $a \in \mathsf{State}{\to}\mathsf{State}{\to}\mathbb{B}$ such that for $x \in \mathsf{Var}$:

$$a.s.t \;=\; ((t.x = s.x + 1) \vee (t.x = 0)) \wedge (t \upharpoonright \{x\}^{\mathsf{c}} = s \upharpoonright \{x\}^{\mathsf{c}})$$

is an action that non-deterministically increases the value of x by 1 or assigns 0 to it. It leaves other variables un-touched. The deterministic action

$$\text{if } a = 0 \text{ then } x := 1 \text{ else skip}$$

from the program Fizb can be represented by $b \in \mathsf{State} \to \mathsf{State} \to \mathbb{B}$ such that:

$$b.s.t \;=\; ((s.a = 0) \Rightarrow (t.x = 1)) \wedge ((s.a \neq 0) \Rightarrow (t.x = s.x)) \wedge (t \upharpoonright \{x\}^{\mathsf{c}} = s \upharpoonright \{x\}^{\mathsf{c}})$$

In general, assignments and conditionals can be defined as:

$$\mathsf{assign}.x.e \;=\; (\lambda s, t.\ (t.x = e.s) \wedge (t \upharpoonright \{x\}^{\mathsf{c}} = s \upharpoonright \{x\}^{\mathsf{c}})) \qquad (3.2.1)$$
$$\text{if } g \text{ then } a \text{ else } b \;=\; (\lambda s, t.\ (g.s \Rightarrow a.s.t) \wedge (\neg g.s \Rightarrow b.s.t)) \qquad (3.2.2)$$

$\mathsf{assign}.x.e$ is usually denoted as $x := e$. Note that the 'expression' e has the type $\mathsf{State} \to \mathsf{Val}$ and the 'boolean expression' g in the conditional has the type $\mathsf{State} \to \mathbb{B}$.

Other examples are skip, miracle, and chaos:

Definition 3.2.1 skip, miracle, and chaos *SKIP, MIRA, CHAOS*

$$\begin{aligned} \mathsf{skip} &= (\lambda s, t.\ s = t) \\ \mathsf{miracle} &= (\lambda s, t.\ \mathsf{false}) \\ \mathsf{chaos} &= (\lambda s, t.\ \mathsf{true}) \end{aligned}$$

◀

The action skip does not change any variable. The action chaos is totally non-deterministic, and is considered to be a very bad action; that is, we do not want to have an action that behaves like chaos. Another kind of actions that we do not want is non-terminating actions. With our choice of representation for actions, we cannot express non-termination. We can still say though, that a non-terminating action will be at least as bad as chaos.[3][4]. An action is therefore said to be potentially non-terminating if it behaves like chaos if executed in some states. The action miracle forbids any transition. It may not even remain in the old state. Such an action is considered to be un-implementable. An action is said to be potentially miraculous if it behaves like miracle if executed in some states. It is called non-miraculous if it is not potentially miraculous. In this thesis non-miraculous actions are called *always enabled* because such an action is always ready to make a transition (even though it may be a skip transition).

Definition 3.2.2 Always Enabled Action *ALWAYS_ENABLED*

$$\square_{\mathsf{En}} a \;=\; (\forall s :: (\exists t :: a.s.t))$$

◀

[3] So, by trying to get rid of chaos, we will automatically get rid of non-termination.

[4] In any case, this should not matter since later, beginning from Chapter 4, all actions considered will be assumed to be terminating.

An action a is refined by, or implemented by, another action b, denoted by $a \sqsubseteq b$, if a can do any transition that b can do. In other words, a is less deterministic than b. The ordering $\sqsubseteq$ corresponds to the reverse of the standard sub-relation ordering, and defines a lattice on Action. The corresponding least upper bound and greatest lower bound operators will be denoted by $\sqcap$ and $\sqcup$.

Definition 3.2.3 REFINEMENT *aREF*

$$a \sqsubseteq b \;=\; (\forall s, t :: b.s.t \Rightarrow a.s.t)$$

Definition 3.2.4 SYNCHRONIZATION OPERATOR *rINTER*

$$a \sqcap b \;=\; (\lambda s, t.\ a.s.t \wedge b.s.t)$$

Definition 3.2.5 CHOICE OPERATOR *rUNION*

$$a \sqcup b \;=\; (\lambda s, t.\ a.s.t \vee b.s.t)$$

◀

chaos is the worst of all actions: it can be refined by any other action. On the other extreme, $\mathsf{miracle}$ refines any other action, which is why it is called "miracle", and why a potentially miraculous action is considered un-implementable. The operator $\sqcap$ is also called *synchronization operator* because it can be used to model a concurrent and synchronized execution of two actions: a transition can be made only if both actions agree on it. The operator reduces non-determinism but may introduce miracles. The operator $\sqcup$ is called *choice operator* because $a \sqcup b$ may behave either as a or b when executed in a state s. The operator adds non-determinism, and may eliminate miracles.

We can lift the projection $\upharpoonright$ to the action level as follows:

Definition 3.2.6 $\upharpoonright$ ON Action *a_Pj_DEF*

$$(a \upharpoonright V).s.t \;=\; a.(s \upharpoonright V).(t \upharpoonright V)$$

◀

For example, one can show that restricting the deterministic assignment $x := x + 1$ to $\{x\}$ yields a non-deterministic action $(\lambda s, t.\ t.x = s.x{+}1)$. Restricting an action does not generally yield something that makes sense. For example, let x and y be distinct variables. Restricting $y := x + y + 1$ to $\{x\}$ yields $(\lambda s, t.\ (\aleph = s.x + \aleph + 1)\ \wedge\ (s \upharpoonright \{x\} = t \upharpoonright \{x\}))$, which is equal to $\mathsf{miracle}$.

Restricting chaos and $\mathsf{miracle}$ yields chaos and $\mathsf{miracle}$ again. What is perhaps more interesting is $\mathsf{skip} \upharpoonright V$. It is an action that preserves the values of all variables in V, but for the rest it behaves like chaos.

Using the above operators on Action we can, for example, rewrite (3.2.1) to:

$$\mathsf{assign}.x.e \;=\; (\lambda s, t.\ t.x = e.s) \sqcap (\mathsf{skip} \upharpoonright \{x\}^{\mathsf{c}}) \qquad (3.2.3)$$

A simultaneous assignment can be defined as:

$$\begin{array}{c} \mathsf{assign}_2.(x,y).(e_1,e_2) \\ = \\ (\lambda s,t.\ t.x = e_1.s)\ \sqcap\ (\lambda s,t.\ t.y = e_2.s)\ \sqcap\ (\mathsf{skip}\restriction\{x,y\}^{\mathsf{c}}) \end{array} \tag{3.2.4}$$

3.2.1 Ignored and Invisible Variables

Parallel programs interact through shared variables, that is, variables which are written by one program, and read by another program. These variables play a central role in our study of parallel composition, presented in Chapter 4. Intuitively, we know what 'shared', 'read', and 'write' variables are. However, careful study is called for and precise definitions, rather than some intuitive notion, of access-modes are required. It is possible to deduce the access-mode of a variable based on syntactic information. For example, by 'looking into its code' we conclude that x and y are the only write variables of the program Fizb in Figure 3.1. Formalizing this would require us to detail the semantics of the programming language being used and falls beyond the scope of this thesis. What we will provide are definitions of variables' access-modes in terms of our semantic domains, which are State and Action.

A set of variables is V *ignored by* an action a, denoted by $V \not\leftarrow a$, if executing the action in any state does not change the values of these variables. Variables in V^{c} *may* however be written by a. The smallest set of variables which may be written by a is the set of variables which are actually written by a. One can show that V is ignored by a if and only if a refines $\mathsf{skip}\restriction V$, which is an action that preserves the values of the variables in V, but for the rest is totally non-deterministic. Consequently, $a \sqcap (\mathsf{skip}\restriction V)$ always ignores V.

Definition 3.2.7 IGNORED-BY *IG_BY_DEF*

$$V \not\leftarrow a \ = \ (\forall s,t :: a.s.t \Rightarrow (s\restriction V = t\restriction V))$$

Theorem 3.2.8 *IG_BY_ADEF*

$$V \not\leftarrow a \ = \ (\mathsf{skip}\restriction V) \sqsubseteq a$$

◀

The notion of ignored-by will be used to express the writability of a variable by a given action. We will now define its counterpart, used to express readability, for which we turn to the diagram in Figure 3.2. A set of variables V is said to be *invisible* to a, denoted by $V \not\twoheadrightarrow a$, if —as suggested by the commuting diagram— the effect of changing the values of the variables in V (by executing the action $\mathsf{skip}\restriction V^{\mathsf{c}}$) in any state s and then followed by the execution of a can be achieved by first executing a and then $\mathsf{skip}\restriction V^{\mathsf{c}}$. In other words, changing the values of the variables in V will not

$$\begin{tikzcd}
s \arrow[r, "a"'] \arrow[d, "\mathsf{skip}\upharpoonright V^{\mathsf{c}}"'] & \bullet \arrow[d, "\mathsf{skip}\upharpoonright V^{\mathsf{c}}"] \\
\bullet \arrow[r, "a"'] & t
\end{tikzcd}$$

Figure 3.2: *The invisibility diagram.*

influence what a can do to the variables outside V. In formula, we can define this notion of invisibility as follows (this definition is due to R. Udink):

$$V \not\twoheadrightarrow a \;=\; (\mathsf{skip}\upharpoonright V^{\mathsf{c}}); a \;\sqsubseteq\; a; (\mathsf{skip}\upharpoonright V^{\mathsf{c}}) \tag{3.2.5}$$

where ";" denotes the sequential composition of relations:

$$(a;b).s.t \;=\; (\exists s' :: a.s.s' \wedge b.s'.t) \tag{3.2.6}$$

Using the above definition, it is possible for an invisible variable to be written. We will insist a slightly stronger definition (because we find it easier for certain proofs) by requiring that if an invisible variable x is written by an action a then a should also be able to 'ignore' it —that is, to behave like skip with respect to x. More precisely:

$$V \not\twoheadrightarrow a \;=\; (\mathsf{skip}\upharpoonright V^{\mathsf{c}}); (a \sqcap \mathsf{skip}\upharpoonright V) \;\sqsubseteq\; a; (\mathsf{skip}\upharpoonright V^{\mathsf{c}}) \tag{3.2.7}$$

Typically though, a write variable of a program is also a read variable. This is the same, as will be explained latter, as saying that an invisible variable is also ignored. If we assume this (we will do so latter in Chapter 4) then the above two definitions are equal. There are several other ways to formulate the notion of invisibility. Below is the definition we use in our HOL package. It is equivalent with (3.2.7) above.

Definition 3.2.9 INVISIBLE-TO *INVI_DEF*

$$V \not\twoheadrightarrow a$$
$$=$$
$$(\forall s,t,s',t' :: (s\upharpoonright V^{\mathsf{c}} = s'\upharpoonright V^{\mathsf{c}}) \wedge (t\upharpoonright V^{\mathsf{c}} = t'\upharpoonright V^{\mathsf{c}}) \wedge (s'\upharpoonright V = t'\upharpoonright V) \wedge a.s.t \;\Rightarrow\; a.s'.t')$$

Another characterization of $\not\twoheadrightarrow$ is given below. It is perhaps less intuitive but useful in later calculations.

Theorem 3.2.10 *INVI_ADEF*

$$V \not\twoheadrightarrow a \;=\; (a \sqsubseteq (a\upharpoonright V^{\mathsf{c}}) \sqcap (\mathsf{skip}\upharpoonright V)) \;\wedge\; (a\upharpoonright V^{\mathsf{c}} \sqsubseteq a)$$

Notice that as corollaries, V is always invisible to $a \upharpoonright V^c$ and $\mathsf{skip} \upharpoonright V$. Below we give the proof of the theorem above.

Proof:

First, we prove the $\Leftarrow$ side. Assume $a \sqsubseteq (a \upharpoonright V^c) \sqcap (\mathsf{skip} \upharpoonright V)$ and $a \upharpoonright V^c \sqsubseteq a$. We derive:

$$\begin{array}{ll}
 & (s \upharpoonright V^c = s' \upharpoonright V^c) \ \wedge \ (t \upharpoonright V^c = t' \upharpoonright V^c) \ \wedge \ (s' \upharpoonright V = t' \upharpoonright V) \ \wedge \ a.s.t \\
\Rightarrow & \{ \ a \text{ refines } a \upharpoonright V^c \ \} \\
 & (s \upharpoonright V^c = s' \upharpoonright V^c) \ \wedge \ (t \upharpoonright V^c = t' \upharpoonright V^c) \ \wedge \ (s' \upharpoonright V = t' \upharpoonright V) \ \wedge \ a.(s \upharpoonright V^c).(t \upharpoonright V^c) \\
\Rightarrow & \{ \ \text{simple calculation} \ \} \\
 & (s' \upharpoonright V = t' \upharpoonright V) \ \wedge \ a.(s' \upharpoonright V^c).(t' \upharpoonright V^c) \\
\Rightarrow & \{ \ (a \upharpoonright V^c) \sqcap (\mathsf{skip} \upharpoonright V) \text{ refines } a \ \} \\
 & a.s'.t'
\end{array}$$

and hence we conclude that $V \not\twoheadrightarrow a$ holds. For $\Rightarrow$ side, assume $V \not\twoheadrightarrow a$. First we prove $a \sqsubseteq (a \upharpoonright V^c) \sqcap (\mathsf{skip} \upharpoonright V)$, and then $a \upharpoonright V^c \sqsubseteq a$. We derive:

$$\begin{array}{ll}
 & a.(s \upharpoonright V^c).(t \upharpoonright V^c) \ \wedge \ (s \upharpoonright V = t \upharpoonright V) \\
= & \{ \upharpoonright \text{COMPOSITION}_{23} \} \\
 & (s \upharpoonright V^c \upharpoonright V^c = s \upharpoonright V^c) \ \wedge \ (t \upharpoonright V^c \upharpoonright V^c = t \upharpoonright V^c) \ \wedge \ (s \upharpoonright V = t \upharpoonright V) \ \wedge \ a.(s \upharpoonright V^c).(t \upharpoonright V^c) \\
\Rightarrow & \{ \ V \not\twoheadrightarrow a \ \} \\
 & a.s.t
\end{array}$$

and hence we conclude that $(a \upharpoonright V^c) \sqcap (\mathsf{skip} \upharpoonright V)$ refines a. For $a \upharpoonright V^c \sqsubseteq a$ we derive:

$$\begin{array}{ll}
 & a.s.t \\
= & \{ \upharpoonright \text{COMPOSITION}_{23} \} \\
 & (s \upharpoonright V^c = s \upharpoonright V^c \upharpoonright V^c) \ \wedge \ (t \upharpoonright V^c = t \upharpoonright V^c \upharpoonright V^c) \ \wedge \ (s \upharpoonright V^c \upharpoonright V = t \upharpoonright V^c \upharpoonright V) \ \wedge \ a.s.t \\
\Rightarrow & \{ \ V \not\twoheadrightarrow a \ \} \\
 & a.(s \upharpoonright V^c).(t \upharpoonright V^c)
\end{array}$$

and hence we conclude that a refines $a \upharpoonright V^c$.

▲

As an example, consider an action $a =$ "$\mathsf{if}\ z\ \mathsf{then}\ x := x + y\ \mathsf{else\ skip}$". Assume that v, x, y, and z are all distinct variables. The action writes to x, and reads from x, y, and z. We have: $\{y, z\} \not\leftarrow a$ and $\{x\}^c \not\leftarrow a$; but not: $\{x, y\} \not\leftarrow a$. We have $\{v\} \not\twoheadrightarrow a$ and $\{x, y, z\}^c \not\twoheadrightarrow a$; but not: $\{v, x\} \not\twoheadrightarrow a$.

$\not\leftarrow$ and $\not\twoheadrightarrow$ satisfy the following properties:

Theorem 3.2.11 $\not\leftarrow$ ANTI-MONOTONICITY *IG_BY_MONO*

$$(V \subseteq W) \wedge (W \not\leftarrow a) \ \Rightarrow \ (V \not\leftarrow a)$$

Theorem 3.2.12 $\not\twoheadrightarrow$ ANTI-MONOTONICITY *INVI_MONO*

$$(V \subseteq W) \wedge (W \not\twoheadrightarrow a) \wedge (W \not\leftarrow a) \ \Rightarrow \ (V \not\leftarrow a)$$

◀

Notation	Meaning	HOL-definition
true	$(\lambda s.\ \mathsf{true})$	`TT_DEF`
false	$(\lambda s.\ \mathsf{false})$	`FF_DEF`
$\neg p$	$(\lambda s.\ \neg p)$	`pNOT_DEF`
$p \Rightarrow q$	$(\lambda s.\ p.s \Rightarrow q.s)$	`pIMP_DEF`
$p \wedge q$	$(\lambda s.\ p.s \wedge q.s)$	`pAND_DEF`
$p \vee q$	$(\lambda s.\ p.s \vee q.s)$	`pAND_DEF`
$(\forall i : W.i : P.i)$	$(\lambda s.\ (\forall i : W.i : P.i.s))$	`RES_qAND`
$(\exists i : W.i : P.i)$	$(\lambda s.\ (\exists i : W.i : P.i.s))$	`RES_qOR`

Note: p and q are predicates over A. The dummy s ranges therefore over A.

Table 3.1: *Overloading of the boolean operators.*

3.3 State-predicates

A predicate over a set A is a function of type $A \to \mathbb{B}$. In particular, we are interested in predicates over State. Such a predicate is called a *state-predicate*. A state-predicate is used to describe a set of states satisfying a certain property; so is:

$$(\lambda s.\ (s.x = s.y + 1)\ \wedge\ n < s.y)$$

a state-predicate describing all states s in which the value of x is the value of y plus 1 and the value of y is greater than n. In practice, the above state-predicate is usually written as:

$$(x = y + 1)\ \wedge\ n < y$$

So, symbols are being overloaded. Other examples of such overloading are:

"$y < x$", "$p \wedge q$", or "$(\exists i : P.i : x.i = 0)$"

which actually denote:

"$(\lambda s.\ s.y < s.x)$", "$(\lambda s.\ p.s \wedge q.s)$", and "$(\lambda s.\ (\exists i : P.i : s.(x.i) = 0))$"

Usually, this kind of overloading does not cause confusion. Later however, there will be occasions where a careful distinction is called for. It will be helpful if the reader is well aware of this double interpretation. To emphasize this, Table 3.1 shows the 'lifted' meaning of the boolean operators. In this thesis, we assume this kind of overloading in all non-HOL formulas —that is, formulas not appearing in the type-writer font. Whenever the meaning of a predicate expression is likely to confuse the reader we will also give the expanded expression, using λ notation as above. In HOL however, as overloading is not possible, separate symbols have to be used to denote the lifted operators.

The set of all state-predicates is denoted with Pred. A state-predicate p is said to *hold* in a state s if $p.s$ holds. A predicate p over A is said to *hold everywhere*, denoted by $[p]$, if $p.s$ holds for all $s \in A$.

Definition 3.3.1 Everywhere Operator *pSEQ_DEF*

$$[p] \; = \; (\forall s :: p.s)$$

◀

Consider again the program Fizb in Figure 3.1. The variables a, x, y are the only variables of Fizb. A predicate $p \in (\{a, x, y\} \to \mathsf{Val}) \to \mathbb{B}$ is a predicate that is 'local' to the program Fizb because it only contains information on the values of a, x, and y. Such a local predicate is useful, for example, in a trivial case, because its validity in a given state cannot be influenced by other programs that do not write to $\{a, x, y\}$. However, since introducing a predicate of this type would require a sub-typing ability in HOL, which is not available, we have to encode them with predicates over State.

A state-predicate p is said to be *confined* by a set of variables V, denoted by $p \in \mathsf{Pred}.V$, if p does not restrict the value of any variable outside V (unless p is already empty):

Definition 3.3.2 Predicate Confinement *CONF_DEF*

$$p \in \mathsf{Pred}.V \; = \; (\forall s, t :: (s \upharpoonright V = t \upharpoonright V) \Rightarrow (p.s = p.t))$$

◀

For example, $x + 1 < y$ is confined by $\{x, y\}$ but not by $\{x\}$. Notice that if p is confined by V, p does not contain useful information about variables outside V. Indeed, $p \in \mathsf{Pred}.V$ is how we encode $p \in (V \to \mathsf{Val}) \to \mathbb{B}$.

true and false are confined by any set. Confinement is preserved by any predicate operator in Table 3.1. So, for example, if $p, q \in \mathsf{Pred}.V$ then $p \wedge q \in \mathsf{Pred}.V$. As a rule of thumb, any predicate p is confined by $\mathsf{free}.p$, that is, the set of variables occurring free in p:

$$p \in \mathsf{Pred}.(\mathsf{free}.p) \tag{3.3.1}$$

Note however, that $\mathsf{free}.p$ is not necessarily the smallest set which confines p. For example, $\emptyset$ confines "$0 = x \vee 0 \neq x$". Another useful property is monotonicity:

Theorem 3.3.3 Confinement Monotonicity *CONF_MONO*

$$V \subseteq W \Rightarrow \mathsf{Pred}.V \subseteq \mathsf{Pred}.W$$

◀

Projection can be lifted to the predicate level:

$$p \upharpoonright V \; = \; (\lambda s.\, p.(s \upharpoonright V)) \tag{3.3.2}$$

Projection distributes over predicate-operators in Table 3.1. So, for example $(p \vee q) \upharpoonright V = (p \upharpoonright V) \vee (q \upharpoonright V)$. Just as in the case of actions, restricting a predicate does in general not yield something that makes sense. For example, given that x and y are distinct variables, restricting $(\lambda s.\ 0 < s.x)$ to $\{y\}$ yields $(\lambda s.\ 0 < \aleph)$ the truth of which at any given state cannot be proved or contradicted since there is nothing we know about $\aleph$, except that it exists. Still, restricting $(\lambda s.\ 0 < s.x)$ to $\{x\}$ yields $(\lambda s.\ 0 < s.x)$ again. In fact, one can show that $p = p \upharpoonright V$ characterizes confinement by V:

$$p \in \mathsf{Pred}.V \;=\; (p = p \upharpoonright V) \tag{3.3.3}$$

3.4 Specification

The effect of an action can be specified by a Hoare triple. If p and q are state-predicates and a is an action, the triple $\{p\}\ a\ \{q\}$ means that if a is executed in any state satisfying p, it will end in a state satisfying q.

Definition 3.4.1 Hoare Triple *HOA_DEF*

$$\{p\}\ a\ \{q\} \;=\; (\forall s,t :: p.s \wedge a.s.t \Rightarrow q.t)$$

◀

Consequently, miracle satisfies $\{p\}$ miracle $\{q\}$ for all state-predicates p and q, whereas chaos only satisfies $\{p\}$ chaos $\{\mathsf{true}\}$ and $\{\mathsf{false}\}$ chaos $\{q\}$. Notice that if a refines b, that is, $b \sqsubseteq a$, then any Hoare triple satisfied by b is also satisfied by a. Basic laws for Hoare triples, such as pre-condition strengthening and post-condition weakening, are well known. See for example [Dij76, Gri81].

An action, if restricted to a set of variables V, cannot be influenced by any variable outside V. That is, if a satisfies $\{p\}\ a\ \{q\}$, then $a \upharpoonright V$ satisfies $\{p \upharpoonright V\}\ a \upharpoonright V\ \{q \upharpoonright V\}$. Consequently, if p and q are confined by V, then $a \upharpoonright V$ also satisfies $\{p\}\ a \upharpoonright V\ \{q\}$.

Theorem 3.4.2 *HOA_Pj_MAP*

$$\frac{\{p\}\ a\ \{q\}}{\{p \upharpoonright V\}\ a \upharpoonright V\ \{q \upharpoonright V\}}$$

Corollary 3.4.3 *HOA_INVI*

$$\frac{V^{\mathsf{c}} \not\rightarrow a \;\wedge\; \{p\}\ a\ \{q\}}{\{p \upharpoonright V\}\ a\ \{q \upharpoonright V\}}$$

▶

The proof of Corollary 3.4.3 is as follows:
Proof:

$$\{p \upharpoonright V\}\ a\ \{q \upharpoonright V\}$$

$\Leftarrow$ { property of refinement }

$$
\begin{array}{ll}
 & a \restriction V \sqsubseteq a \;\wedge\; \{p \restriction V\}\; a \restriction V\; \{q \restriction V\} \\
\Leftarrow & \{\text{ Theorem 3.4.2 }\} \\
 & a \restriction V \sqsubseteq a \;\wedge\; \{p\}\; a\; \{q\} \\
\Leftarrow & \{\text{ Theorem } 3.2.10_{27} \} \\
 & V^{\mathsf{c}} \not\twoheadrightarrow a \;\wedge\; \{p\}\; a\; \{q\}
\end{array}
$$

▲

Sometimes actions are extended with a concurrent assignment to some fresh variables. This is done either to simplify a proof or as a part of some program transformation. If a is an action and b is an assignment to some fresh variables, we can encode the concurrent extension of a with b by $b; a$. Recall that ";" is an action-level composition. That is, $b; a$ is considered to be a single atomic action rather than two separate actions which have to be executed in some order. Extending an action with a concurrent assignment to some fresh variables 'preserves' the specification of the action. In fact, this is a corollary of the above theorems:

Corollary 3.4.4 ADDING FRESH VARIABLES *HOA_ADD_FRESH*

$$\frac{V \not\leftarrow b \;\wedge\; V^{\mathsf{c}} \not\twoheadrightarrow a \;\wedge\; \{p\}\; a\; \{q\}}{\{p \restriction V\}\; b; a\; \{q \restriction V\}}$$

▶

In the above, the fact that b is an assignment to some variables which are not referred to by a is expressed by $V \not\leftarrow b \;\wedge\; V^{\mathsf{c}} \not\twoheadrightarrow a$. Notice that if both p and q are confined by V then $\{p\}\; b\; \{q\}$ is really preserved by $a; b$. The proof of the above is given below.

Proof:

We will use the following well-known rule of action composition:

$$\frac{\{p\}\; a\; \{q\} \;\wedge\; \{q\}\; a\; \{r\}}{\{p\}\; a\; \{r\}} \tag{3.4.1}$$

Now, we derive:

$$
\begin{array}{ll}
 & \{p \restriction V\}\; b; a\; \{q \restriction V\} \\
\Leftarrow & \{\text{ the above rule of action composition }\} \\
 & (\{p \restriction V\}\; b\; \{p \restriction V\}) \;\wedge\; (\{p \restriction V\}\; a\; \{q \restriction V\}) \\
\Leftarrow & \{\; V \not\leftarrow b, \text{ Theorem } 3.2.8_{26} \} \\
 & (\{p \restriction V\}\; (\mathsf{skip} \restriction V)\; \{p \restriction V\}) \;\wedge\; (\{p \restriction V\}\; a\; \{q \restriction V\}) \\
\Leftarrow & \{\text{ Theorem } 3.4.2_{31} \text{ and Corollary } 3.4.3_{31} \} \\
 & (\{p\}\; a\; \{q\}) \;\wedge\; V^{\mathsf{c}} \not\twoheadrightarrow a
\end{array}
$$

▲

Let a be an action and V be a set of variables such that a may write only to variables in V. That is, $V^{\mathsf{c}} \not\leftarrow a$ holds. Let p be a state-predicate confined by a set of variables W. Suppose a is executed in a state satisfying p, and suppose in its execution a does not change any variable in $V \cap W$. Then a will maintain p. The action a may still change the values of the variables in $V \backslash W$, but this does not destroy p since p is confined by W, which means that p tolerates changes in the variables in W^c.

Theorem 3.4.5 `Action_Wr_Pred`

$$\frac{V^c \not\leftarrow a \;\wedge\; p \in \mathsf{Pred}.W}{\{(\lambda s.\ s{\restriction}U = f{\restriction}U) \wedge p\}\ a\ \{(\lambda s.\ s{\restriction}U = f{\restriction}U) \Rightarrow p\}} \quad \text{let } U = V \cap W$$

Proof:

$$\begin{array}{ll}
 & (s{\restriction}U = f{\restriction}U) \;\wedge\; p.s \;\wedge\; a.s.t \;\wedge\; (t{\restriction}U = f{\restriction}U) \\
\Rightarrow & \{ \text{ Theorem } 3.2.8_{26}\text{: } V^c \not\leftarrow a \;=\; \mathsf{skip}{\restriction}V^c \sqsubseteq a \ \} \\
 & (s{\restriction}U = f{\restriction}U) \;\wedge\; p.s \;\wedge\; (s{\restriction}V^c = t{\restriction}V^c) \;\wedge\; (t{\restriction}U = f{\restriction}U) \\
\Rightarrow & \{ \ W\backslash U \subseteq V^c \text{ and } {\restriction}\,\textsc{Anti-monotonicity}_{23} \ \} \\
 & (s{\restriction}U = t{\restriction}U) \;\wedge\; p.s \;\wedge\; (s{\restriction}(W\backslash U) = t{\restriction}(W\backslash U)) \\
\Rightarrow & \{ \ {\restriction}\,\textsc{Extension}_{23} \text{ by } \cup \ \} \\
 & (s{\restriction}W = t{\restriction}W) \;\wedge\; p.s \\
\Rightarrow & \{ \ p \in \mathsf{Pred}.W\text{, hence } p = p{\restriction}W \ \} \\
 & p.t
\end{array}$$

▲

The following corollary of the theorem above should be familiar to the reader. It states that if p is confined by V and a does not write V, then a cannot destroy p.

Corollary 3.4.6 `Action_Wr_Pred_Cor`

$$\frac{V \not\leftarrow a \;\wedge\; p \in \mathsf{Pred}.V}{\{p\}\ a\ \{p\}}$$

3.5 Summary

Just to provide the reader with an overview: we have in this chapter revealed how we represent variables, program states, predicates, actions, and specifications. We have discussed how the notion of read and write variables can be encoded. All these issues are relevant to answer the question (which will be answered in the next chapter) as to how we represent a program. In addition, many useful theorems presented in this chapter will be used to derive various results regarding program transformation and composition presented in the next chapter.

3.6 Related Work

In Action Systems [Bac90] there exists also a notion of projection on states and predicates, but the topic seems to receive little attention. In [Cho93] projection and coercion of predicates are used to relate programs with different state spaces. [dBKPJ93] extensively discusses a semantics and proof rules for variable hiding (a hidden variable, also called local variable, is a variable written by a program but is ignored by and invisible to its environment). Introducing local variables is useful as progress made on local variables cannot be influenced by other programs. In [UHK94] projection is used to encode the concept of local predicates. Program properties based on local invariants are defined and their preservation under parallel composition is studied.

"Thea! No! What are you doing?"
"I'm going to let the titans in."
The amulet flared brightly,
The gate swung open ...
–from Into the Labyrinth

Chapter 4 Programs and Their Properties

A brief review on the programming logic UNITY will be presented. Special attention will be given to the issue of compositionality, that is, the ability to decompose a specification of a program into specifications of its components. A new progress operator which is more compositional will be introduced, and a set of calculational laws for the new operator will be provided.

4.1 Introduction

THE previous chapter introduced the basic building blocks which constitute a program, namely variables and actions. In this chapter we will talk about programs and their behavior. Basically, a program is only a collection of actions. During its execution, the actions are executed in a certain order. It is however possible to encode the ordering in the actions themselves by adding program counters. In this sense, a program is really a collection of actions, without any ordering. This way of viewing programs is especially attractive if we consider a parallel execution of actions where strict orderings begin to break down. In fact, a number of distributed programming logics are based on this idea. Examples thereof are Action Systems [Bac90], Temporal Logic of Action [Lam90], and UNITY [CM88].

Recall that we aim to verify the work of Lentfert [Len93] on a general, self-stabilizing, and distributed algorithm to compute minimal distances in a hierarchical network. Lentfert's work is based on the programming logic UNITY, which was chosen mainly for its simplicity and its high level of abstraction. We will simply follow his choice.

Examples of programs derivation and verification using UNITY are many. The introductory book to UNITY [CM88] itself contains numerous examples, ranging from a simple producer-consumer program, to a parallel garbage collection program. Realistic problems have also been addressed. In [Sta93] Staskauskas derives an I/O sub-system of an existing operating system, which is responsible for allocating I/O resources. In [Piz92] Pizzarello used UNITY to correct an error found in a large operating system. The fault had been corrected before, and verified using the traditional approach of testing and tracing [KB87]. It is interesting to note that the amount of work using UNITY is small, compared to that of the traditional approach. A review of Pizzarello's industrial experience on the use of UNITY can be found in [Piz91]. In [CKW$^+$91] Chakravarty and his colleagues developed a simulation of the diffusion and aggregation of particles in a cement like porous media.

In practice, many useful programs do not, in principle, terminate; some examples are file servers, routing programs in computer networks, and control systems in an air plane. For such a program, its responses during its execution are far more important than the state it ends up with when it eventually terminates. To specify such a program we cannot therefore use Hoare triples as we did in Chapter 3. Two aspects are especially important: *progress* and *safety*. A progress property of a program expresses what the program is expected to eventually realize. For example, if a message is sent through a routing system, a progress property may state that eventually the message will be delivered to its destination. A safety property, on the other hand, tells us what the program should not do: for example, that the message is only to be delivered to its destination, and not to any other computer. The two kinds of properties are not mutually exclusive. For example, a safety property, stating that a computer in a network should *not* either ignore an incoming message or discard it, implies that the computer should either consume the message or re-route it to some neighbors. This states progress. In UNITY there is an operator called *unless* to express safety properties, and two more operators, *ensures* and *leads-to*, to express progress properties.

In [CM88] two kinds of program composition are discussed. In the first, programs are composed by simply 'merging' them. This can be thought of as modelling parallel composition. In the second, called *super-position*, actions may be extended with concurrent assignments to fresh variables. Both will be discussed here, but in a quite different light than in [CM88]. In addition we will discuss program compositions in which guards may be added. In the applications verified in this thesis however, only parallel composition is used.

Ideally, a program is developed hand in hand with its proof. One starts with a specification, which is refined, step by step using a set of rules, until an implementable program is obtained. In sequential programming, one begins with an imaginary program. In each development step, the program is refined by adding more details to it. The original specification is then reduced to specifications for the components of the programs. Subsequently, each component can be developed in isolation. This kind of hierarchical program decomposition is not an issue which is very well explored in UNITY. Safety properties decompose nicely. Unfortunately, the same cannot be said for progress properties, especially with respect to the parallel composition. A general law to decompose progress is provided by Singh [Sin89]. Similar laws were also used by Lentfert to decompose self-stabilization. An unpleasant discovery that was made during our research was that these laws were all flawed[1]. Fortunately, the flaw was not so serious that it bears no consequence to Lentfert's results. To facilitate the mechanical verification of Lentfert's work, we have also re-written Lentfert's proofs as to make them simpler and more intuitive. This is made possible by, among other things, the Transparency Law (2.2.4). If the reader recalls the discussion in Section 2.2, the law is used to assign the task of establishing a progress property to a write-disjoint

[1] Partly, the discovery is due to the absolute rigor imposed by HOL. When a supposedly obvious fact seems to be impossible to be proven in HOL, it is a good indication that we do not formulate the fact correctly and completely.

```
prog   Fizban
read   $\{a, x, y\}$
write  $\{x, y\}$
init   true
assign
       if $a = 0$ then $x := 1$ else skip
[]     if $a \neq 0$ then $x := 1$ else skip
[]     if $x \neq 0$ then $y, x := y + 1, 0$ else skip
```

Figure 4.1: *The program* Fizban

component. For the purpose of this law, a new progress operator will be introduced.

Section 4.2 briefly reviews the ideas behind UNITY. Section 4.3 discusses the standard UNITY operators to express behavior of a program. Various basic laws used to reason about them are presented in Section 4.4. Section 4.5 introduces the new progress operator mentioned above. Various laws about parallel composition will be discussed in Section 4.7. Section 4.8 discusses the parallel composition of write-disjoint programs. Section 4.9 briefly discusses some other kinds of program composition. And finally, Section 4.10 briefly discusses the soundness of UNITY with respect to its operational semantics.

4.2 UNITY Programs

UNITY is a programming logic invented by Chandy and Misra in 1988 [CM88] for reasoning about safety and progress behavior of distributed programs. Figure 4.1 displays an example. The precise syntax will be given later.

The **read** and **write** sections declare, respectively, the read and write variables of the program[2]. The **init** section describes the assumed initial states of the program. In the program Fizban in Figure 4.1, the initial condition is **true**, which means that the program may start in any state. The **assign** section lists the actions of the programs, separated by the [] symbol.

The actions in a UNITY program are assumed to be *atomic*. An execution of a UNITY program is an infinite and interleaved execution of its actions. In a fully parallel system, each action may be thought of as being executed by a separate processor. To make our reasoning independent from the relative speed of the processors, nothing is said about when a particular action should be executed. Consequently, there is no ordering imposed on the execution of the actions. There is a *fairness* condition though:

[2] When declaring a variable one may also want to declare its type (instead of assuming all variables to be of type, say, $\mathbb{N}$). UNITY does not disallow such a declaration, but its logic as in [CM88] does not include laws to deal with subtleties which may arise from typing the variables. For example, nothing is said about the effect of assigning the number 10 to a $\mathbb{B}$ valued variable. Of course it is possible to extend UNITY with a type theory, but this issue is beyond the scope of this thesis.

in a UNITY execution, which is infinite, each action must be executed infinitely often (and hence cannot be ignored forever).

For example, by now the reader should be able to guess that in the program Fizban, eventually $x = 0$ holds and that if $M = y$, then eventually $M < y$ holds.

Notice that the program Fizban resembles the program Fizb in Figure 3.1_{22}, which has the following body:

```
do forever
    begin
    if $a = 0$ then $x := 1$ else skip ;
    if $a \neq 0$ then $x := 1$ else skip ;
    if $x \neq 0$ then $y, x := y + 1, 0$ else skip
    end
```

In fact, Fizb is a sequential implementation of Fizban. Indeed, as far as UNITY concerns, the actions can be implemented sequentially, fully parallel, or anything in between, as long as the atomicity and the fairness conditions of UNITY are being met. Perhaps, the best way to formulate the UNITY's philosophy is as worded by Chandy and Misra in [CM88]:

> *A UNITY program describes* ***what*** *should be done in the sense that it specifies the initial state and the state transformations (i.e., the assignments). A UNITY program does not specify precisely* ***when*** *an assignment should be executed ... Neither does a UNITY program specify* ***where*** *(i.e., on which processor in a multiprocessor system) an assignment is to be executed, nor to which process an assignment belongs.*

That is, in UNITY one is encouraged to concentrate on the 'real' problem, and not to worry about the actions ordering and allocation, as such are considered to be implementation issues.

Despite its simple view, UNITY has a relatively powerful logic. The wide range of applications considered in [CM88] illustrates this fact quite well. Still, to facilitate programming, more structuring methods would be appreciated. An example thereof is sequential composition of actions. Structuring is an issue which deserves more investigation in UNITY.

By now the reader should have guessed that a UNITY program P can be represented by a quadruple (A, J, V_r, V_w) where $A \subseteq$ Action is a set consisting of the actions of P, $J \in$ Pred is a predicate describing the possible initial states of P, and $V_r, V_w \in$ Var are sets consisting of respectively read and write variables of P. The set of all such structures will be denoted by Uprog. So, all UNITY programs will be a member of this set, although, as will be made clear later, the converse is not necessarily true.

To access each component of an Uprog object, the destructors **a**, **ini**, **r**, and **w** are introduced. They satisfy the following property:

Theorem 4.2.1 Uprog DESTRUCTORS *ID_UPROG*

$$P \in \mathsf{Uprog} \;=\; (P \;=\; (\mathbf{a}P, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P))$$

In addition, the input variables of P, that is, the variables read by P but not written by it, is denoted by $\mathbf{i}P$:

$$\mathbf{i}P = \mathbf{r}P \backslash \mathbf{w}P \tag{4.2.1}$$

4.2.1 The Programming Language

Below is the syntax of UNITY programs that is used in this thesis. The syntax deviates slightly from the one in [CM88][3].

$$\begin{array}{lll} \langle \textit{Unity Program} \rangle & ::= & \mathsf{prog}\ \langle \textit{name of program} \rangle \\ & & \mathsf{read}\ \langle \textit{set of variables} \rangle \\ & & \mathsf{write}\ \langle \textit{set of variables} \rangle \\ & & \mathsf{init}\ \langle \textit{predicate} \rangle \\ & & \mathsf{assign}\ \langle \textit{actions} \rangle \end{array}$$

actions is a list of *action*s separated by $[\!]$. An *action* is either a single action or a set of indexed actions.

$$\begin{array}{lll} \langle \textit{actions} \rangle & ::= & \langle \textit{action} \rangle \mid \langle \textit{action} \rangle\ [\!]\ \langle \textit{actions} \rangle \\ \langle \textit{action} \rangle & ::= & \langle \textit{single action} \rangle \mid ([\!] i : i \in V : \langle \textit{actions} \rangle_i) \end{array}$$

A single action is either a simple assignment such as $x := x + 1$ or a guarded action. A simple assignment can simultaneously assign to several variables. An example is $x, y := y, x$ which swaps the values of x and y. The precise meaning of assignments has been given in Chapter 3. A guarded action has the form:

$$\begin{array}{l} \mathsf{if}\ g_1\ \mathsf{then}\ a_1 \\ \quad g_2\ \mathsf{then}\ a_2 \\ \quad g_3\ \mathsf{then}\ a_3 \\ \quad \ldots \end{array}$$

An else part can be added with the usual meaning. If more than one guard is true then one is selected non-deterministically. If none of the guards is true, a guarded action behaves like skip. So, for example, the action "if $a \neq 0$ then $x := 1$ else skip" from the program Fizban can also be written as "if $a \neq 0$ then $x := 1$".

In addition we have the following requirements regarding the well-formedness of a UNITY program:

i. A program has at least one action.

[3] We omit the always section and we find it necessary to split the declare section into read and write parts

ii. The actions of a program should only write to the declared write variables.

iii. The actions of a program should only depend on the declared read variables.

iv. A write variable is also readable.

These are perfectly natural requirements for a program. Most programs that a programmer writes will satisfy them[4].

In Chapter 3 the notions of ignored and invisible variables have been explained. If a set of variables V^{c} is ignored by an action a, that is, $V^{\mathsf{c}} \not\leftarrow a$, then a can only write to the variables in V. So, ***ii*** can be encoded as:

$$(\forall a : a \in \mathbf{a}P : (\mathbf{w}P)^{\mathsf{c}} \not\leftarrow a)$$

Quite similarly, If V^{c} is invisible to a, that is, $V^{\mathsf{c}} \not\twoheadrightarrow a$, then a will only depend on the variables in V. So, ***iii*** can be encoded as:

$$(\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\twoheadrightarrow a)$$

In [CM88] it is required that all actions in a UNITY program are deterministic. We find that this restriction is unnecessary. If a program contains a non-deterministic action, the only consequence is that the program will probably show less predictable behavior. In [CM88] it is also required that all actions in a UNITY program are terminating. This is a perfectly logical requirement because if a statement does not terminate, then no further progress will be made, which violates the infinite execution model of UNITY. Here, we will refrain from imposing this requirement. We consider non-termination to be as bad as chaos, which is totally non-deterministic. A program which contains chaos can always be refined by removing some non-determinism at the action level. We leave it to the designers to come up with actions which are terminating.

The requirement that a UNITY guarded action behaves as skip if none of its guards is true means that all UNITY actions are required to be always-enabled (= not potentially miraculous). It is also not necessary to require this explicitly, but we will do it anyway. As we will see later, this requirement is crucial for a law called Impossibility Law.

Recall that any UNITY program is an object of type Uprog. Now we can define a predicate Unity to define the well-formedness of an Uprog object. From now on, with a "UNITY program", we mean an object satisfying Unity.

Definition 4.2.2 Unity *UNITY*

$$\begin{aligned}\mathsf{Unity}.P \;=\;& (\mathbf{a}P \neq \emptyset) \;\wedge\; (\mathbf{w}P \subseteq \mathbf{r}P) \;\wedge\; (\forall a : a \in \mathbf{a}P : \Box_{\mathsf{En}} a) \;\wedge \\ & (\forall a : a \in \mathbf{a}P : (\mathbf{w}P)^{\mathsf{c}} \not\leftarrow a) \;\wedge\; (\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\twoheadrightarrow a)\end{aligned}$$

◀

[4] One may however want to drop requirements ***i*** and ***iv***. ***i*** is required by some laws. So, if omitted it will re-appear somewhere else. ***iv*** was added because it seems convenient.

4.2.2 Parallel Composition

A consequence of the absence of ordering in the execution of a UNITY program is that the parallel composition of two programs can be modelled by simply merging the variables and actions of both programs. In UNITY parallel composition is denoted by $[\!]$. In [CM88] the operator is also called *program union*.

Definition 4.2.3 PARALLEL COMPOSITION *PAR*

$$P[\!]Q \;=\; (\mathbf{a}P \cup \mathbf{a}Q, \mathbf{ini}P \wedge \mathbf{ini}Q, \mathbf{r}P \cup \mathbf{r}Q, \mathbf{w}P \cup \mathbf{w}Q)$$

◄

Parallel composition is reflexive, commutative, and associative. It has a unit element, namely $(\emptyset, \mathsf{true}, \emptyset, \emptyset)$ (although this is not a well-formed UNITY program).

As an example, we can compose the program Fizban in Figure 4.1 in parallel with the program below:

```
prog    TikTak
read    {$a$}
write   {$x$}
init    true
assign  if $a = 0$ then $a := 1$ [] if $a \neq 0$ then $a := 0$
```

The resulting program consists of the following actions (the else skip part of the actions in Fizban will be dropped, which is, as remarked in Section 4.2.1, allowed):

```
$a_0$ :  if $a = 0$ then $a := 1$
$a_1$ :  if $a \neq 0$ then $a := 0$
$a_2$ :  if $a = 0$ then $x := 1$
$a_3$ :  if $a \neq 0$ then $x := 1$
$a_4$ :  if $x \neq 0$ then $y, x := y + 1, 0$
```

Whereas in Fizban $x \neq 0$ will always hold somewhere in the future, the same cannot be said for Fizban $[\!]$ TikTak. Consider the execution sequence $(a_0; a_2; a_1; a_3; a_4)*$, which is a fair execution and therefore a UNITY execution. In this execution, the assignment $x := 1$ will never be executed. If initially $x \neq 1$ this will remain so for the rest of this execution sequence.

4.3 Programs' Behavior

To facilitate reasoning about program behavior UNITY provides several primitive operators. The discussion in Section 4.2 revealed that an execution of a UNITY program never, in principle, terminates. Therefore we are going to focus on the behavior of a program *during* its execution. Two aspects will be considered: safety and progress. Safety behavior can be described by an operator called unless. By the fairness condition of UNITY, an action cannot be continually ignored. Once executed, it may

$$p \text{ unless } q : \begin{tikzcd} p \wedge \neg q \arrow[loop left] \arrow[r] & q \end{tikzcd} \qquad p \text{ ensures } q : \begin{tikzcd} p \wedge \neg q \arrow[loop left] \arrow[r, "!"] & q \end{tikzcd}$$

Figure 4.2: unless *and* ensures. *The predicates $p \wedge \neg q$ and q define sets of states. The arrows depict possible transitions between the two sets of states. The arrow marked with ! is a guaranteed transition.*

induce some progress. For example, the execution of the action a_4 in Fizban $[\!]$ TikTak will establish $x = 0$ regardless when it is executed. Actually, any action which is not skip or chaos induces some meaningful progress. This kind of single-action progress is described by an operator called ensures.

In the sequel, P, Q, and R will range over UNITY programs; a, b, and c over Action; and p, q, r, s, J and K over Pred.

Definition 4.3.1 UNLESS *UNLESS*

$$_P\vdash p \text{ unless } q \;=\; (\forall a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{p \vee q\})$$

Definition 4.3.2 ENSURES *ENSURES*

$$_P\vdash p \text{ ensures } q \;=\; (\,_P\vdash p \text{ unless } q) \;\wedge\; (\exists a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{q\})$$

Intuitively, $_P\vdash$ p unless q implies that once p holds during an execution of P, it remains to hold at least until q holds. Figure 4.2 may be helpful. Note that this given interpretation says nothing about what p unless q means if p never holds during an execution. $_P\vdash$ p ensures q encompasses p unless q, and adds that there should also exist an action that can, and because of the fairness assumption of UNITY, will establish q.

If progress can be made from p to q, and from q to r, we would expect that progress can also be made from p to r. Similarly, if progress can be made from p_1 to q_1, and from p_2 to q_2, then starting from either p_1 or p_2, progress will be made to either q_1 or q_2. These two are natural properties of progress. However, ensures does not have these properties. It is because it only describes single-action progress. To describe the combined effect of several actions, the smallest transitive and disjunctive closure of ensures has to be used. This relation is denoted by $\mapsto$ ("leads-to"). Leads-to describes progress in general.

The notion of transitivity is well known; we will write Trans.R to denote that a relation R is transitive. The notion of left-disjunctivity is defined below:

Definition 4.3.3 LEFT DISJUNCTIVE RELATION *LDISJ_DEF*

A relation R over $A \to \mathbb{B}$ is called *left-disjunctive*, denoted $\mathsf{Ldisj}.R$ iff for all $q \in A \to \mathbb{B}$ and all non-empty sets W (of predicates over A):

$$(\forall p : p \in W : R.p.q) \;\Rightarrow\; R.(\exists p : p \in W : p).q$$

◀

The formula above may be confusing. Notice that the $\exists$ on the right hand side of $\Rightarrow$ denotes $\exists$ on the predicate level, not Boolean level. If we write the formula without notational overloading (as warned in Section 3.3), it looks like:

$$(\forall p : p \in W : R.p.q) \;\Rightarrow\; R.(\lambda s.\ (\exists p : p \in W : p.s)).q$$

In a simple case, $\mathsf{Ldisj}.R$ implies $R.p_1.q \wedge R.p_2.q \Rightarrow R.(p_1 \vee p_2).q$. For example, unless is left-disjunctive. Now we can define TDC, the smallest transitive and disjunctive closure of a given relation, as follows:

Definition 4.3.4 SMALLEST TRANSITIVE AND DISJUNCTIVE CLOSURE *TDC*

$$\mathsf{TDC}.R.p.q \;=\; (\forall S : R \subseteq S \wedge \mathsf{Trans}.S \wedge \mathsf{Ldisj}.S : S.p.q)$$

◀

Note that we can also define TDC as follows:

$$\mathsf{TDC}.R \;=\; \cap\{S \mid R \subseteq S \wedge \mathsf{Trans}.S \wedge \mathsf{Ldisj}.S\} \tag{4.3.1}$$

which shows more clearly that TDC is some smallest closure of R. Note that the total relation is transitive and left-disjunctive. Hence the set $\{S \mid R \subseteq S \wedge \mathsf{Trans}.S \wedge \mathsf{Ldisj}.S\}$ is non-empty, and hence $\mathsf{TDC}.R$ is non-trivial.

Now we can define $\mapsto$ as follows:

Definition 4.3.5 LEADS-TO *LEADSTO*

$$(\lambda p, q.\ {}_P\!\vdash p \mapsto q) \;=\; \mathsf{TDC}.(\lambda p, q.\ {}_P\!\vdash p \ \mathsf{ensures}\ q)$$

◀

Introducing TDC may seem only to serve as adding flavor to the notation, but it is not. Many useful properties of $\mapsto$ are actually pure properties of TDC. Later, we introduce a variant of $\mapsto$. The new progress operator is also a TDC-relation, but based on a different relation. This new operator will then automatically inherit all properties of TDC. This has saved a lot of time and effort in mechanically verifying the properties of the new operator.

As an example, the program Fizban in Figure 4.1, which has the following assign section:

```
   if $a = 0$ then $x := 1$
[] if $a \neq 0$ then $x := 1$
[] if $x \neq 0$ then $y, x := y + 1, 0$
```

satisfies the following properties:

$$_{\mathsf{Fizban}}\vdash \quad (a = X)\ \mathsf{unless\ false} \tag{4.3.2}$$
$$_{\mathsf{Fizban}}\vdash \quad \mathsf{true\ unless}\ (x = 1) \tag{4.3.3}$$
$$_{\mathsf{Fizban}}\vdash \quad (a = 0)\ \mathsf{ensures}\ (x = 1) \tag{4.3.4}$$
$$_{\mathsf{Fizban}}\vdash \quad (a \neq 0)\ \mathsf{ensures}\ (x = 1) \tag{4.3.5}$$
$$_{\mathsf{Fizban}}\vdash \quad \mathsf{true} \mapsto (x = 1) \tag{4.3.6}$$

If (4.3.2) holds for any X then it states that Fizban cannot change the value of a. (4.3.3) is an example of a property that trivially holds in any program. The reader can check it by unfolding the definition of unless.

(4.3.4) and (4.3.5) describe single-action progress from, respectively $a = 0$ and $a \neq 0$ to $x = 1$. Because $\mapsto$ is a closure of ensures, and hence includes ensures, we conclude that $(a = 0) \mapsto (x = 1)$ and $(a \neq 0) \mapsto (x = 1)$ also hold. Using the disjunctivity property of progress, we can conclude (4.3.6), which states that eventually $x = 1$. Note that ensures is *not* disjunctive. So, despite (4.3.4) and (4.3.5), we cannot conclude:

$$_{\mathsf{Fizban}}\vdash \mathsf{true\ ensures}\ (x = 1) \tag{4.3.7}$$

The above cannot be true because there is no single action in Fizban which can establish $x = 1$ regardless in which state it is executed. The point is that single-action progress is a bit special and cannot be expressed using $\mapsto$. The composition Fizban $[\!]$ TikTak, despite property (4.3.6), does not satisfy $\mathsf{true} \mapsto (x = 1)$. On the other hand, composing TikTak with a program P that does satisfy (4.3.7) yields a program that does satisfy $\mathsf{true} \mapsto (x = 1)$.

Properties of the form $_P\vdash p$ unless false are called *stable* properties, which are very useful properties because they express that once p holds during any execution of P, it will remain to hold. Because of their importance we will define a separate abbreviation:

Definition 4.3.6 STABLE PREDICATE *STABLE*

$$_P\vdash \circlearrowright p \;=\; _P\vdash p\ \mathsf{unless\ false}$$

$_P\vdash \circlearrowright p$ is pronounced "*p is stable in P*" and p is called a *stable predicate*. Notice that $\circlearrowright$ can also be defined as follows:

$$_P\vdash \circlearrowright p \;=\; (\forall a : a \in \mathsf{a}P : \{p\}\ a\ \{p\}) \tag{4.3.8}$$

Consequently, if p holds initially and is stable in P, it will hold throughout any execution of P, and hence it is an *invariant*. There seem to be at least two notions of invariant. Here, we define an invariant of a program P as a predicate that holds throughout any execution of P. Note that with this definition, an invariant is not necessarily stable. For example, consider a program P consisting of a single action a:

```
if $x = 1$ then $x := 2$
```

If $\mathbf{ini}P = (x = 0)$, then $(x = 0) \vee (x = 1)$ holds initially and throughout the execution of P. Hence it is an invariant. However, $(x = 0) \vee (x = 1)$ is *not* stable (because if $x = 1$ then a will assign 2 to x).

Invariants and stable predicates are disjunctive and conjunctive. That is, if p and q are invariants (or stable), then so are $p \wedge q$ and $p \vee q$. However, whereas invariants are monotonic with respect to $\Rightarrow$, stable predicates are not. Invariants are useful, but for self-stabilizing programs, whose initial condition can be as liberal as true, the notion of stability is more useful.

4.4 UNITY Laws

Figures 4.3 and 4.4 display a set of basic laws for unless properties. Figure 4.5 displays a set of basic laws for $\circlearrowright$, and 4.6 for ensures. The properties are taken from [CM88]. As a notational convention: *if it is clear from the context which program P is meant, we often omit it from a formula.* For example we may write p unless q to mean ${}_P\vdash p$ unless q. Also, for laws we write, for example:

$$P : \frac{\ldots(p \ \mathsf{unless}\ q)\ldots}{r \mapsto s} \quad \text{to abbreviate:} \quad \frac{\ldots({}_P\vdash p \ \mathsf{unless}\ q)\ldots}{{}_P\vdash r \rightarrowtail s}$$

Note that the unless Conjunction and Disjunction laws in Figure 4.3 can be generalized to combine an arbitrary number of unless properties (the generalization has also been verified). A similar remark also holds for the conjunction and disjunction of $\circlearrowright$. Note that ensures is conjunctive but *not* disjunctive. Note also that the ensures Introduction law depends on the fact that the program P is non-empty (otherwise there is no sense in talking about 'single-action' progress).

There is another law of ensures called Impossibility Law, stating that it is impossible to progress to false unless if one starts from false, which is just not possible:

Theorem 4.4.16 ensures Impossibility *ENSURES_IMPOS*

$$P : \frac{p \ \mathsf{ensures\ false}}{[\neg p]}$$

A crucial assumption to this law is that all actions in the program P are always-enabled. Otherwise, if a miracle is possible, then it can be used to establish false, and then it would follow by the ensures Post-weakening law in page 47 that any progress is possible. The proof of the Impossibility law is as follows:

Proof:

By definition, ${}_P\vdash p$ ensures false implies that there exists an action $a \in \mathbf{a}P$ such that $\{p\}\ a\ \{\mathsf{false}\}$ holds. Since P is a UNITY program, a is always enabled. We derive:

$\{p\}\ a\ \{\mathsf{false}\}$

$=$ { definition Hoare triple }

Theorem 4.4.1 unless INTRODUCTION *UNLESS_IMP_LIFT1, UNLESS_IMP_LIFT2*

$$P: \frac{[p \Rightarrow q] \ \vee \ [\neg p \Rightarrow q]}{p \text{ unless } q}$$

Theorem 4.4.2 unless POST-WEAKENING *UNLESS_CONSQ_WEAK*

$$P: \frac{(p \text{ unless } q) \ \wedge \ [q \Rightarrow r]}{p \text{ unless } r}$$

Theorem 4.4.3 unless CONJUNCTION *UNLESS_CONJ*

$$P: \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{p \wedge r \text{ unless } (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

Theorem 4.4.4 unless DISJUNCTION *UNLESS_DISJ*

$$P: \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{p \vee r \text{ unless } (\neg p \wedge s) \vee (\neg r \wedge q) \vee (q \wedge s)}$$

◄

Figure 4.3: *Basic laws for* unless.

Corollary 4.4.5 unless REFLEXIVITY *UNLESS_REFL*

p unless p

Corollary 4.4.6 ANTI-REFLEXIVITY *UNLESS_ANTI_REFL*

$\neg p$ unless p

Corollary 4.4.7 SIMPLE CONJUNCTION *UNLESS_SIMPLE_CONJ*

$$P: \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{p \wedge r \text{ unless } q \vee s}$$

Corollary 4.4.8 SIMPLE DISJUNCTION *UNLESS_SIMPLE_DISJ*

$$P: \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{p \vee r \text{ unless } q \vee s}$$

◄

Figure 4.4: *Some corollaries of* unless.

Theorem 4.4.9 $\circlearrowright$ CONJUNCTION *STABLE_GEN_CONJ*

$$P : \frac{(\circlearrowright p) \;\wedge\; (\circlearrowright q)}{\circlearrowright (p \wedge q)}$$

Theorem 4.4.10 $\circlearrowright$ DISJUNCTION *STABLE_GEN_DISJ*

$$P : \frac{(\circlearrowright p) \;\wedge\; (\circlearrowright q)}{\circlearrowright (p \vee q)}$$

◀

Figure 4.5: *Basic laws of* $\circlearrowright$.

Theorem 4.4.11 ensures INTRODUCTION *ENSURES_IMP_LIFT*

$$P : \frac{[p \Rightarrow q]}{p \textsf{ ensures } q}$$

Theorem 4.4.12 ensures POST-WEAKENING *ENSURES_CONSQ_WEAK*

$$P : \frac{(p \textsf{ ensures } q) \;\wedge\; [q \Rightarrow r]}{p \textsf{ ensures } r}$$

Theorem 4.4.13 ensures PROGRESS SAFETY PROGRESS (PSP) *ENSURES_PSP*

$$P : \frac{(p \textsf{ unless } q) \;\wedge\; (r \textsf{ unless } s)}{p \wedge r \textsf{ ensures } (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

◀

Figure 4.6: *Basic laws for* ensures

Corollary 4.4.14 ensures REFLEXIVITY *ENSURES_REFL*

p ensures p

Corollary 4.4.15 ensures CONJUNCTION *ENSURES_CONJ*

$$P : \frac{(p \textsf{ ensures } q) \;\wedge\; (r \textsf{ ensures } s)}{p \wedge r \textsf{ ensures } (p \wedge s) \vee (r \wedge q) \vee (q \wedge s)}$$

◀

Figure 4.7: *Some corollaries of* ensures

$$
\begin{array}{ll}
 & (\forall s,t :: p.s \wedge a.s.t \Rightarrow \mathsf{false}) \\
= & \quad \{ \text{ predicate calculus } \} \\
 & (\forall s,t :: p.s \Rightarrow \neg a.s.t) \\
= & \quad \{ \ a \text{ is always enabled, that is, } (\forall s :: (\exists t :: a.s.t)) \ \} \\
 & (\forall s :: \neg p.s) \\
= & \quad \{ \text{ definition of } [.] \ \} \\
 & [\neg p]
\end{array}
$$

▲

4.4.1 Transitive and Disjunctive Relations

Recall that the general progress operator $\mapsto$ is defined as the TDC of ensures. That is, it is the smallest transitive and left-disjunctive closure of ensures. Many laws of leads-to can be derived from general properties of TDC, which are presented in this subsection.

Being the smallest closure of some sort, TDC induces an induction principle. Actually it is trivial: since TDC.R is the smallest transitive and left-disjunctive closure of R, any other relation S which includes R, is transitive, and left-disjunctive will also include TDC.R.

Theorem 4.4.17 TDC Induction *TDC_INDUCT1*

$$\frac{R \subseteq S \ \wedge \ \mathsf{Trans}.S \ \wedge \ \mathsf{Ldisj}.S}{\mathsf{TDC}.R \subseteq S}$$

◀

The principle gives a sufficient condition for a relation S to include TDC.R. In [CM88] the principle is invoked many times to prove other laws for $\mapsto$.

It can be shown that TDC.R itself includes R, and is transitive and left-disjunctive. These last two are the most basic properties progress. In addition, it is also monotonic with respect to $\subseteq$.

Theorem 4.4.18 *TDC_LIFT, TDC_TRANS, TDC_LDISJ*

$$(R \subseteq \mathsf{TDC}.R) \ \wedge \ \mathsf{Trans}.(\mathsf{TDC}.R) \ \wedge \ \mathsf{Ldisj}.(\mathsf{TDC}.R)$$

Theorem 4.4.19 TDC Monotonicity *TDC_MONO*

$$(R \subseteq S) \ \Rightarrow \ (\mathsf{TDC}.R \subseteq \mathsf{TDC}.S)$$

◀

In [CM88] some laws of the form $\mapsto \subseteq S$ are proven through $\mapsto \subseteq (\mapsto \cap S)$. In general, to show TDC.$R \subseteq$ (TDC.$R \cap S$), using TDC Induction it suffices to show:

$$(R \subseteq \mathsf{TDC}.(\mathsf{TDC}.R \cap S)) \ \wedge \ \mathsf{Trans}.(\mathsf{TDC}.R \cap S) \ \wedge \ \mathsf{Ldisj}.(\mathsf{TDC}.R \cap S) \qquad (4.4.1)$$

The above is often easier to prove. Note that part of it is by Theorem 4.4.18 trivial[5]. We have defined $\mapsto$ as the smallest transitive and left-disjunctive closure of ensures. One may wonder if replacing "transitive" with "right-transitive" would still define the same relation. That is, we would like to define $\mapsto$ as:

$$\mapsto \ = \ \cap\{S \mid (\,\mathsf{ensures}\ \subseteq S) \wedge (\mathsf{ensures}\,; S \subseteq S) \wedge \mathsf{Ldisj}.S\} \tag{4.4.2}$$

Note that $R; S \subseteq S$ means that S is right-transitive with respect to the base relation R. The question whether or not the above definition of $\mapsto$ is equal to the old one can be generalized to the question whether or not the following equation holds:

$$\mathsf{TDC}.R \ = \ \cap\{S \mid R \subseteq S \wedge (R; S \subseteq S) \wedge \mathsf{Ldisj}.S\} \tag{4.4.3}$$

Let $\Sigma^{\blacktriangleright}.R$ abbreviate the set $\{S \mid R \subseteq S \wedge (R; S \subseteq S) \wedge \mathsf{Ldisj}.S\}$ and let $\mathsf{TDC}^{\blacktriangleright}.R$ abbreviate $\cap(\Sigma^{\blacktriangleright}.R)$. Analogous to the case with TDC, one can show that $\mathsf{TDC}^{\blacktriangleright}.R$ itself is a closure of R, is right-transitive, and left-disjunctive. That is, $\mathsf{TDC}^{\blacktriangleright}.R$ is a member of $\Sigma^{\blacktriangleright}.R$. Being the smallest closure, $\mathsf{TDC}^{\blacktriangleright}$ also induces an induction principle:

$$\frac{R \subseteq S \ \wedge \ (R; S \subseteq S) \ \wedge \ \mathsf{Ldisj}.S}{\mathsf{TDC}^{\blacktriangleright}.R \subseteq S} \tag{4.4.4}$$

This, and TDC Induction state sufficient conditions for TDC to be equal to $\mathsf{TDC}^{\blacktriangleright}$. Among these conditions, the only non-trivial one is that $\mathsf{TDC}^{\blacktriangleright}.R$ is transitive, but this can be proven using the induction principle (4.4.4). The conclusion is that TDC and $\mathsf{TDC}^{\blacktriangleright}$ are equal:

Theorem 4.4.20 *TDC_EQU_Ri_TDC*

$$\mathsf{TDC} \ = \ \mathsf{TDC}^{\blacktriangleright}$$

◀

As a corollary, the definition (4.4.2) of $\mapsto$ is equal to the old one. It also means the following induction principle is applicable to $\mapsto$:

$$P: \ \frac{(\mathsf{ensures} \subseteq S) \ \wedge \ (\mathsf{ensures}; S \subseteq S) \ \wedge \ \mathsf{Ldisj}.S}{\mapsto \subseteq S} \tag{4.4.5}$$

There are cases where the induction principle above is more useful than the one in Theorem 4.4.17. For example, it has been crucial in proving a progress law called Completion law (page 52).

Another kind of induction that is often used in practice is well-founded induction. A relation $\prec \in A \to A \to \mathbb{B}$ is said to be *well-founded* if it is not possible to construct an infinite sequence of ever decreasing values in A. That is, $\ldots, x_2 \prec x_1 \prec x_0$ is

[5] For example, to prove $\mathsf{Trans}.(\mathsf{TDC}.R \cap S)$ we have to prove:

$$\mathsf{TDC}.R.p.q \wedge S.p.q \wedge \mathsf{TDC}.R.q.r \wedge S.q.r \ \Rightarrow \ \mathsf{TDC}.R.p.q \wedge S.p.r$$

for all p, q, and r. However, since by by Theorem 4.4.18 $\mathsf{TDC}.R$ is already transitive, we only need to show $S.p.r$.

not possible. For example the ordering $<$ on $\mathbb{N}$ and $\subset$ on sets and relations are well-founded. A well-founded relation admits the well-founded induction principle given below[6].

Definition 4.4.21 Well-founded Induction *ADMIT_WF_INDUCTION*

A relation $\prec \in A \to A \to \mathbb{B}$ is said to admit the well-founded induction if:

$$(\forall y :: (\forall x : x \prec y : p.x) \Rightarrow p.y) \;=\; (\forall y :: p.y)$$

◄

Let m be a function —so-called bound function— that maps State to A. If from p a program can progress to q, or else it maintains p while decreasing the value of m with respect to a well-founded ordering $\prec$, then, since $\prec$ is well-founded, it is not possible to keep decreasing m, and hence eventually q will be established. We call this principle *Bounded Progress*. It holds for any relation on predicates which is reflexive, transitive, and left-disjunctive.

Let in the sequel $\to$ be a relation over predicates over B (which can be program states), $\prec$ be a *well founded* relation over a *non-empty* type A, and m be a mapping from B to A.

Theorem 4.4.22 Bounded Progress *BOUNDED_REACH_i*

$$\frac{\begin{array}{c}\mathsf{Trans.} \to \;\wedge\; \mathsf{Ldisj.} \to \;\wedge\; q \to q \\ (\forall M :: p \wedge (m = M) \to (p \wedge (m \prec M)) \vee q)\end{array}}{p \to q}$$

►

Note: The notation is overloaded. In the above, "$p\wedge(m = M)$" and "$(p\wedge(m \prec M))\vee q$" actually mean, respectively, $(\lambda s.\ p.s \wedge (m.s = M))$ and $(\lambda s.\ (p.s \wedge (m.s \prec M)) \vee q.s)$

Proof:

$$\begin{array}{ll}
 & p \to q \\
\Leftarrow & \{ \to \text{ is left-disjunctive and } B \text{ is non-empty } \} \\
 & (\forall M :: p \wedge (m = M) \to q) \\
= & \{ \text{Well-Founded Induction} \} \\
 & (\forall M :: (\forall M' : M' \prec M : p \wedge (m = M') \to q) \;\Rightarrow\; (p \wedge (m = M) \to q))
\end{array}$$

If M is a minimal element, thus there is no M' such that $M' \prec M$, then the assumption:

$$p \wedge (m = M) \to (p \wedge (m \prec M)) \vee q$$

is equal to $p \wedge (m = M) \to q$, which trivially implies the last formula in the derivation above. If M is not a minimal element:

[6] It has been showed that the above formulation of well-foundedness is actually equivalent with the admittance of the well-founded induction itself.

$$
\begin{array}{ll}
 & (\forall M' : M' \prec M : p \wedge (m = M') \rightarrow q) \\
\Rightarrow & \{ \rightarrow \text{ is left-disjunctive } \} \\
 & p \wedge (m \prec M) \rightarrow q \\
\Rightarrow & \{ q \rightarrow q;\ \rightarrow \text{ is left-disjunctive } \} \\
 & (p \wedge (m \prec M)) \vee q \rightarrow q \\
\Rightarrow & \{ \text{from the assumption: } p \wedge (m = M) \rightarrow (p \wedge (m \prec M)) \vee q;\ \rightarrow \text{ is transitive } \} \\
 & p \wedge (m = M) \rightarrow q
\end{array}
$$

▲

A corollary of the Bounded Progress principle is the following, stating that if from $\neg p$ progress can be made in which the value of the bound function m decreases, then eventually p will be reached.

Theorem 4.4.23 Inevitable Fulfilment *BOUNDED_ALWAYS_REACH_i*

$$
\frac{\mathsf{Trans.} \rightarrow \ \wedge\ \mathsf{Ldisj.} \rightarrow \ \wedge\ q \rightarrow q \qquad (\forall M :: \neg p \wedge (m = M) \rightarrow (m \prec M))}{\mathsf{true} \rightarrow p}
$$

◀

4.4.2 Laws of Leads-to

We are not going to use the $\mapsto$ operator very often. A variant thereof, better suited for our purpose, will be introduced in Section 4.5. For the sake of completeness, Figure 4.8 displays a set of basic laws for $\mapsto$. Some of them follow directly from the laws mentioned in the previous subsection. The reader may want to compare those laws with those of the new operator given in Section 4.5.

4.5 Introducing the Reach Operator

Consider the program P and TikToe in Figure 4.9. The program P can establish $x = 1$ if a is less than 2. So, it satisfies $_P\vdash (a < 2) \mapsto (x = 1)$. As with Hoare triples, we may strengthen the 'pre-condition' of $\mapsto$, and come up with the following property of P:

$$(b = 0) \wedge (a < 2) \mapsto (x = 1) \qquad (4.5.1)$$

We note that P does not write to either a or b, and hence should maintain $(b = 0) \wedge (a < 2)$. We expect then, that if we compose P in parallel with another program which maintains the stability of $(b = 0) \wedge (a < 2)$, but does not write to x —the program TikToe is an example thereof— then (4.5.1) will be respected by the composition. At least, this works with $P [] \mathsf{TikToe}$. The programs P and TikToe are *write-disjoint*, that is, they do not write to a common write variable. Compositions of write-disjoint

Theorem 4.4.24 $\mapsto$ INTRODUCTION *LEADSTO_IMPLICATION, LEADSTO_ENS_LIFT_thm*

$$P: \frac{[p \Rightarrow q] \ \vee \ (p \ \mathsf{ensures}\ q)}{p \mapsto q}$$

Theorem 4.4.25 $\mapsto$ INDUCTION *LEADSTO_INDUCT_thm1*

$$P: \frac{(\forall p, q :: (p \ \mathsf{ensures}\ q) \Rightarrow R.p.q) \ \wedge \ \mathsf{Trans}.R \ \wedge \ \mathsf{Ldisj}.R}{(\forall p, q :: (p \mapsto q) \Rightarrow R.p.q)}$$

Theorem 4.4.26 $\mapsto$ TRANSITIVITY *LEADSTO_TRANS_thm*

$$P: \frac{(p \mapsto q) \ \wedge \ (q \mapsto r)}{p \mapsto r}$$

Theorem 4.4.27 $\mapsto$ DISJUNCTION *LEADSTO_GEN_DISJ*
For all non-empty sets W:

$$P: \frac{(\forall i : i \in W : p.i \mapsto q.i)}{(\exists i : i \in W : p.i) \mapsto (\exists i : i \in W : q.i)}$$

Corollary 4.4.28 $\mapsto$ CANCELLATION *LEADSTO_CANCEL*

$$P: \frac{(p \mapsto q \vee b) \ \wedge \ (b \mapsto r)}{p \mapsto q \vee r}$$

Corollary 4.4.29 $\mapsto$ STRENGTHENING & WEAKENING *LEADSTO_ANTE_STRONG, LEADSTO_CONSQ_WEAK*

$$P: \frac{[p \Rightarrow q] \ \wedge \ (q \mapsto r) \ \wedge \ [r \Rightarrow s]}{p \mapsto s}$$

Theorem 4.4.30 $\mapsto$ PROGRESS SAFETY PROGRESS (PSP) *LEADSTO_PSP*

$$P: \frac{(p \mapsto q) \ \wedge \ (r \ \mathsf{unless}\ s)}{p \wedge r \mapsto (q \wedge r) \vee s}$$

Theorem 4.4.31 $\mapsto$ IMPOSSIBILITY *LEADSTO_IMPOS*

$$P: \frac{p \mapsto \mathsf{false}}{[\neg p]}$$

Theorem 4.4.32 $\mapsto$ COMPLETION *LEADSTO_COMPLETION*
For all *finite* sets W:

$$P: \frac{(\forall i : i \in W : p.i \mapsto q.i \vee r) \ \wedge \ (\forall i : i \in W : q.i \ \mathsf{unless}\ r)}{(\forall i : i \in W : p.i) \mapsto (\forall i : i \in W : q.i) \vee r}$$

◀

Figure 4.8: *Basic laws of* $\mapsto$

```
prog    $P$                               prog    TikToe
read    $\{a, x\}$                        read    $\{a, b\}$
write   $\{x\}$                           write   $\{a\}$
init    true                              init    true
assign  if $a < 2$ then $x := 1$          assign  if $a = 0$ then $a := 1$
[]      if $a = 2$ then $x := x + 1$      []      if $a = 1$ then $a := 0$
                                          []      if $b \neq 0$ then $a := a + 1$
```

Figure 4.9: *The program P and* TikToe.

programs are frequently found in practice. In Chapter 2 we hypothesize that the parallel composition of write-disjoint programs satisfies a principle called Transparency Principle. It states that progress made through the writable part of a write-disjoint component P will be preserved by the composition, as long as the other write-disjoint components respect whatever assumptions P has on its non-writable part. The example with P and TikToe suggests that the principle is valid. It is valid, as we will see later, but not if we use $\mapsto$ to specify progress.

To illustrate the problem with $\mapsto$ consider the following program P', with the same read and write variables as P, and the same initial condition. However, the action if $a < 2$ then $x := 1$ in P will be broken in two. P' has the following assign section:

```
   if $a = 0$ then $x := 1$
[] if $a = 1$ then $x := 1$
[] if $a = 2$ then $x := x + 1$
```

The program P' also satisfies (4.5.1). However, if composed with TikToe the progress may fail if both programs choose a wrong order of execution. Consider the following scheduling of the actions in $P' [] $TikToe:

```
[ if $a = 0$ then $a := 1$ ; if $a = 0$ then $x := 1$ ; if $b \neq 0$ then $a := a + 1$ ;
  if $a = 1$ then $a := 0$ ; if $a = 1$ then $x := 1$ ; if $a = 2$ then $x := x + 1$ ]*
```

which is fair, but if initially all a, b, and x are 0, then x will never become 1 in this execution sequence.

So, if we imagine P as a program which is still under development (so, we cannot look into its code), and if the specification of P states that it should satisfy $(b = 0) \wedge (a < 2) \mapsto (x = 1)$ and $\{a, b\} \not\subseteq \mathbf{w}P$, we cannot just say that composing it with TikToe will preserve $(b = 0) \wedge (a < 2) \mapsto (x = 1)$. The moral of the story is that we cannot generally conclude the $\mapsto$ properties of a composite program from the $\mapsto$ properties of its components, without further information about the interior of the components, or at least, information about how the components' properties are derived. There are however cases where it is possible. A sufficient condition was given by Singh in [Sin89]. This will be discussed in Section 4.7. But even the result by Sigh is not strong enough to derive the Transparency Principle. In this section, a new progress operator will be introduced, with which the principle is provable.

Let us define the following variant of ensures:

Definition 4.5.1 $\mathfrak{ensures}$ *B_ENS*

$$J\ {}_P\!\vdash p\ \mathfrak{ensures}\ q\ =\ p,q \in \mathsf{Pred}.(\mathbf{w}P)\ \wedge\ ({}_P\!\vdash \circlearrowleft J) \wedge ({}_P\!\vdash J \wedge p\ \mathsf{ensures}\ q)$$

◀

Note that $\mathfrak{ensures}$ only describes progress through the writable part of a program. An additional parameter J is added, which is required to be stable in the program. Since the state of the non-writable part of the program cannot change, it can be specified within J.

A variant of leads-to called "reach", denoted by $\rightarrowtail$, can be defined as the smallest transitive and left-disjunctive closure of $\mathfrak{ensures}$. It follows that $\rightarrowtail$ can only specify progress made through the writable part of a program, but this should not be a hindrance as such is the only kind of progress a program can make.

Definition 4.5.2 Reach *REACH*

$$(\lambda p,q.\ J\ {}_P\!\vdash p \rightarrowtail q)\ =\ \mathsf{TDC}.(\lambda p,q.\ J\ {}_P\!\vdash p\ \mathfrak{ensures}\ q)$$

◀

Alternatively, we can also define $\rightarrowtail$ as follows:

$$\begin{array}{c} J\ {}_P\!\vdash p \rightarrowtail q \\ = \\ ({}_P\!\vdash \circlearrowleft J)\ \wedge\ \mathsf{TDC}.(\lambda r,s.\ ({}_P\!\vdash J \wedge r\ \mathsf{ensures}\ s) \wedge r,s \in \mathsf{Pred}.(\mathbf{w}P)).p.q \end{array} \tag{4.5.2}$$

It should now be obvious why we introduced TDC. The properties of TDC mentioned in Subsection 4.4.1 can easily be instantiated for $\rightarrowtail$. This operator is *not* equal to $\mapsto$, but let us postpone the details until Section 4.10. It is however easy to see that:

$$(\mathsf{true}\ {}_P\!\vdash p \rightarrowtail q) \Rightarrow ({}_P\!\vdash p \mapsto q) \tag{4.5.3}$$

By its definition, $\mathsf{true}\ {}_P\!\vdash p\ \mathfrak{ensures}\ q$ implies ${}_P\!\vdash p\ \mathsf{ensures}\ q$. Hence, by TDC Monotonicity$_{48}$, (4.5.3) follows.

As a notational convention: *if it is clear from the context which program P or which stable predicate J are meant, we often omit them from an expression.* For example we may write ${}_P\!\vdash p \rightarrowtail q$ or even simply $p \rightarrowtail q$ to mean $J\ {}_P\!\vdash p \rightarrowtail q$. Also, for laws we write, for example:

$$P,J:\ \frac{\ldots(p\ \mathsf{unless}\ q)\ldots}{r \rightarrowtail s}\quad \text{to abbreviate:}\quad \frac{\ldots({}_P\!\vdash p\ \mathsf{unless}\ q)\ldots}{J\ {}_P\!\vdash r \rightarrowtail s}$$

Figure 4.11 displays a set of basic laws of $\rightarrowtail$ which have corresponding laws for $\mapsto$. The proofs of these properties follow the pattern of the related proofs for $\mapsto$ properties as found in [CM88]. Figure 4.12 displays some laws which have no $\mapsto$ counterpart. Some comment as to how the laws can be proven is also included. In particular, the $\rightarrowtail$ Confinement law states that an expression of the form $J\ {}_P\!\vdash p \rightarrowtail q$ is only valid if both

```
prog    Buffer
read    {in, inRdy, ack, buf, out}
write   {ack, buf, out}
init    ¬ack ∧ (buf = [])
assign
        if inRdy ∧ ¬ack ∧ (ℓ.buf < N) then buf, ack := buf ++[in], true
[]      if ¬inRdy ∧ ack then ack := false
[]      if buf ≠ [] then out, buf := hd.buf, tl.buf
```

Figure 4.10: *An N-placed buffer.*

p and q are confined by $\mathsf{Pred}.(\mathbf{w}P)$. This confirms what is said before, namely that $\rightarrowtail$ describes only progress through the writable-part of a program. The $\rightarrowtail$ Disjunction states the disjunctivity property of $\rightarrowtail$. It should be noted however, that $\rightarrowtail$ is *not* disjunctive in its J-argument. That is, $J_1 \vdash p \rightarrowtail q$ and $J_2 \vdash p \rightarrowtail q$ do not necessarily imply $J_1 \wedge J_2 \ \vdash p \rightarrowtail q$. There are good reasons for this, but let us postpone the discussion for a while.

As an example, consider the program Buffer displayed in Figure 4.10. It uses a buffer buf of size N. What it does is passing on the values in buf to out, first in, first out. Data are entered to buf via the input register in. The boolean variable inRdy is an input variable, which is expected to become true if a new datum becomes available. Buffer is ready to receive a new datum if there is a place in buf and if ack is false. When a new datum is entered to buf, it is acknowledged by setting ack true. A property of the program Buffer is that a new and un-acknowledged datum will eventually appear in out. Using $\mapsto$ this can be expressed as follows:

$$(\forall X :: \ _{\mathsf{Buffer}}\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \mapsto (\mathsf{out} = X)) \tag{4.5.4}$$

Using $\rightarrowtail$ the property can be expressed as follows:

$$(\forall X :: (\mathsf{in} = X) \wedge \mathsf{inRdy} \ _{\mathsf{Buffer}}\vdash \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X)) \tag{4.5.5}$$

However, the following, which would be quite tempting to write due to its resemblance to (4.5.4):

$$(\forall X :: \mathsf{true} \ _{\mathsf{Buffer}}\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X)) \tag{4.5.6}$$

is *not* a valid expression because the argument "$(\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack}$" is not a predicate confined by $\mathbf{w}(\mathsf{Buffer})$.

The previously mentioned Transparency Law will be presented and proven in Section 4.7.

4.6 On the Substitution Law

Recall that we call a predicate p invariant in a program P if it holds throughout any computation of P. Consequently, the truth of p can be assumed in manipulating specifications of P, if by a 'specification' we mean a predicate over all possible executions of

Theorem 4.5.3 $\rightarrowtail$ INTRODUCTION *REACH_ENS_LIFT,REACH_IMP_LIFT*

$$P, J : \frac{\begin{array}{c} p, q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ (\circlearrowright J) \\ [J \wedge p \Rightarrow q] \ \vee \ (J \wedge p \ \mathsf{ensures} \ q) \end{array}}{p \rightarrowtail q}$$

Theorem 4.5.4 $\rightarrowtail$ INDUCTION *REACH_INDUCT1*

$$P, J : \frac{\begin{array}{c} (\forall p, q :: (p \ \mathfrak{ensures} \ q) \Rightarrow R.p.q) \\ \mathsf{Trans}.R \ \wedge \ \mathsf{Ldisj}.R \end{array}}{(p \rightarrowtail q) \Rightarrow R.p.q}$$

Theorem 4.5.5 $\rightarrowtail$ TRANSITIVITY *REACH_TRANS*

$$P, J : \frac{(p \rightarrowtail q) \ \wedge \ (q \rightarrowtail r)}{p \rightarrowtail r}$$

Theorem 4.5.6 $\rightarrowtail$ DISJUNCTION *REACH_DISJ*
For all *non-empty* sets W:

$$P, J : \frac{(\forall i : i \in W : p.i \rightarrowtail q.i)}{(\exists i : i \in W : p.i) \rightarrowtail (\exists i : i \in W : q.i)}$$

Corollary 4.5.7 $\rightarrowtail$ REFLEXIVITY *REACH_REFL*

$$P, J : \frac{p \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ (\circlearrowright J)}{p \rightarrowtail p}$$

Corollary 4.5.8 $\rightarrowtail$ CANCELLATION *REACH_CANCEL*

$$P, J : \frac{q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ (p \rightarrowtail q \vee r) \ \wedge \ (r \rightarrowtail s)}{p \rightarrowtail q \vee s}$$

Theorem 4.5.9 $\rightarrowtail$ PSP *REACH_PSP*

$$P, J : \frac{r, s \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ (r \wedge J \ \mathsf{unless} \ s) \wedge \ (p \rightarrowtail q)}{p \wedge r \rightarrowtail (q \wedge r) \vee s}$$

Theorem 4.5.10 $\rightarrowtail$ COMPLETION *REACH_COMPLETION*
For all *finite* and *non-empty* sets W:

$$P, J : \frac{\begin{array}{c} r \in \mathsf{Pred}.(\mathbf{w}P) \\ (\forall i : i \in W : q.i \wedge J \ \mathsf{unless} \ r) \ \wedge \ (\forall i : i \in W : p.i \rightarrowtail q.i \vee r) \end{array}}{(\forall i : i \in W : p.i) \rightarrowtail (\forall i : i \in W : q.i) \vee r}$$

◀

Figure 4.11: *Basic properties of* $\rightarrowtail$ *which are analogous to those of* $\mapsto$

Theorem 4.5.11 $\rightarrowtail$ STABLE SHIFT *REACH_STABLE_SHIFT*

$$P: \frac{p_2 \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\circlearrowleft J) \ \wedge \ (J \wedge p_2 \vdash p_1 \rightarrowtail q)}{J \vdash p_1 \wedge p_2 \rightarrowtail q}$$

Can be proven using $\rightarrowtail$ INDUCTION.

Theorem 4.5.12 $\rightarrowtail$ STABLE STRENGTHENING *REACH_STAB_MONO*

$$P: \frac{(\circlearrowleft J_2) \ \wedge \ (J_1 \vdash p \rightarrowtail q)}{J_1 \wedge J_2 \vdash p \rightarrowtail q}$$

Can be proven using $\rightarrowtail$ INDUCTION.

Corollary 4.5.13 $\rightarrowtail$ STABLE BACKGROUND *REACH_IMP_STABLE*

$$P: \frac{J \vdash p \rightarrowtail q}{\circlearrowleft J}$$

Follows straightforwardly from the alternative definition of $\rightarrowtail$ (4.5.2).

Theorem 4.5.14 $\rightarrowtail$ CONFINEMENT *REACH_IMP_CONF*

$$P, J: \frac{p \rightarrowtail q}{p, q \in \mathsf{Pred.}(\mathbf{w}P)}$$

Can be proven using $\rightarrowtail$ INDUCTION.

Theorem 4.5.15 $\rightarrowtail$ SUBSTITUTION *REACH_SUBST*

$$P, J: \frac{\begin{array}{c} p, s \in \mathsf{Pred.}(\mathbf{w}P) \\ [J \wedge p \Rightarrow q] \ \wedge \ (q \rightarrowtail r) \ \wedge \ [J \wedge r \Rightarrow s] \end{array}}{p \rightarrowtail s}$$

Follows from $\rightarrowtail$ INTRODUCTION, STABLE BACKGROUND, TRANSITIVITY, and CONFINEMENT.

Figure 4.12: *More properties of* $\rightarrowtail$

P. This very natural principle is imposed in [CM88] as an axiom called the *Substitution Law*, and has the following form. Let $\mathfrak{R}$ be either unless, ensures, or $\mapsto$:

$$P: \frac{\begin{array}{c} \textit{"J is invariant"} \\ [J \Rightarrow (p = p')] \ \wedge \ \mathfrak{R}.p.q \ \wedge \ [J \Rightarrow (q = q')] \end{array}}{\mathfrak{R}.p'.q'} \qquad (4.6.1)$$

The law was a source of anxiety because it was found that the law makes the logic inconsistent [San91]. On the other hand, without the Substitution Law UNITY is incomplete relative to a certain operational semantics. In [San91] Sanders proposed an extension, from which the law can be derived instead of imposed as an axiom. The consistency of the logic was thus guaranteed.

In this section we will briefly discuss the Substitution Law and Sander's extension, and their relation to the UNITY logic plus the $\rightarrowtail$ operator as we have described so far.

As we have no Substitution Law, at least, not for **unless**, **ensures**, and $\mapsto$, one may ask what kind of incompleteness one may expect. Let us consider just the case of unless. The other two operators can be argued about in much the same way. Recall that in Section 4.3 we have carefully given the following operational interpretation for unless:

> *Intuitively,* $_P\vdash p$ unless q **implies** *that once p holds during an execution of P, it remains to hold at least until q holds.*

This deviates slightly from the traditional interpretation, for example as in [CM88], in which the "implies" above is replaced by "if and only if". Indeed, we use "implies" because we wish to avoid questions about incompleteness, until now. Now let us see what kind of problem we run into if we replace "implies" with "if and only if".

Let $_P\vdash p\,\mathfrak{U}\,q$ means "once p holds during an execution of P, it remains to hold at least until q holds". Now consider a program Lazy as follows:

```
prog    Lazy
read    $\{a, x\}$
write   $\{x\}$
init    $(a \neq 0) \wedge (x = 0)$
assign  if $a = 0$ then $x := x + 1$
```

In Lazy we have $(x = 0)$ unless $(a = 0)$. Because initially $a \neq 0$, the value of x will remain constant. So, $(x = 0)\,\mathfrak{U}$ false holds. However, $(x = 0)$ unless false cannot be derived without the Substitution Law:

$$\begin{array}{ll} & (x = 0) \text{ unless } (a = 0) \\ = & \quad \{\ a \neq 0 \text{ is invariant, Substitution Law }\} \\ & (x = 0) \text{ unless false} \end{array}$$

But $_{\mathsf{Lazy}}\vdash (x = 0)$ unless false by the definition of unless cannot hold since it is equal to false. So obviously, unless is *not* equal to the interpretation $\mathfrak{U}$[7].

In [San91] Sanders introduced a variant unless which is equal to $\mathfrak{U}$ (and similarly for ensures and $\mapsto$). Let $_P\vdash \Box J$ means that J is invariant. Sanders defines unless_S as follows:

$$_P\vdash p \text{ unless}_S\ q \;=\; (\exists J : {_P\vdash} \Box J : J \wedge p \text{ unless } q) \qquad (4.6.2)$$

For J, one can always choose the strongest invariant of P. This strongest invariant characterizes all states reachable by P, which is why the Substitution Law is derivable

[7] This is caused by the fact that unless is defined in terms of *all* states instead of those states which are actually reachable from the starting state(s).

from the definition above. The unless as defined above coincides with the interpretation $\mathfrak{U}$

To explicitly record on which invariant a property is based, Sanders generalized (4.6.2) by parameterizing unless_S with an invariant:

$$J\ {}_P\!\vdash p\ \mathsf{unless}_S\ q\ =\ (\ {}_P\!\vdash \Box J) \wedge (\ {}_P\!\vdash (J \wedge p)\ \mathsf{unless}\ q) \tag{4.6.3}$$

Curiously, Sanders fell into the same trap as Chandy and Misra did in [CM88] by claiming that the Substitution Law also applies to the parameterized unless_S. Consider again the program Lazy. Note that $x < 2$ is invariant. We derive now:

$$\begin{array}{ll}
 & \mathsf{true} \\
= & \{\ (4.6.3),\ x<2 \text{ is invariant, } \mathsf{unless}\ \textsc{Anti-reflexivity}_{46}\ \} \\
 & (x<2)\ {}_{\mathsf{Lazy}}\!\vdash (x<2)\ \mathsf{unless}_S\ (2 \le x) \\
= & \{\ x<2 \text{ is invariant, Substitution Law }\} \\
 & (x<2)\ {}_{\mathsf{Lazy}}\!\vdash (x<2)\ \mathsf{unless}_S\ \mathsf{false} \\
\Rightarrow & \{\ (4.6.3)\ \} \\
 & {}_{\mathsf{Lazy}}\!\vdash (x<2)\ \mathsf{unless\ false} \\
= & \{\ \text{definition of } \mathsf{Lazy}\ \} \\
 & \mathsf{false}
\end{array}$$

The flaw is corrected by Prasetya in [Pra94] by requiring that J is not only an invariant, but also a 'strong' invariant. A strong invariant is an invariant which is also stable.

Definition 4.6.1 STRONG INVARIANT *Inv*

$$ {}_P\!\vdash \boxed{\mathsf{s}}\, J\ =\ [\mathbf{ini}P \Rightarrow J] \wedge (\ {}_P\!\vdash \circlearrowleft J)$$

Definition 4.6.2 PARAMETERIZED unless_S *UNL*

$$J\ {}_P\!\vdash p\ \mathsf{unless}_S\ q\ =\ (\ {}_P\!\vdash \boxed{\mathsf{s}}\, J) \wedge (\ {}_P\!\vdash J \wedge p\ \mathsf{unless}\ q \wedge J)$$

◀

One can show that the following Substitution Law holds for the above definition of unless_S. The result extends to ensures and $\mapsto$. It has been mechanically verified and available as part of our UNITY package for the theorem prover HOL.

Theorem 4.6.3 unless SUBSTITUTION *UNL_SUBST*

$$P, J:\ \frac{[J \Rightarrow (p = p')]\ \wedge\ (p\ \mathsf{unless}_S\ q)\ \wedge\ [J \Rightarrow (q = q')]}{p'\ \mathsf{unless}_S\ q'}$$

◀

The reader may note that the $\rightarrowtail$ operator already fulfils a $\textsc{Substitution}_{57}$ law. Indeed, the operator has some flavor of Sanders' parameterized $\mapsto$, but there are some important differences. Sanders' invariant-parameterized $\mapsto$ is defined as follows:

$$J\ {}_P\!\vdash p \overset{\mathsf{S}}{\mapsto} q\ =\ (\ {}_P\!\vdash \boxed{\mathsf{s}}\, J)\ \wedge\ (\ {}_P\!\vdash J \wedge p \mapsto q) \tag{4.6.4}$$

Theorem 4.7.1 unless COMPOSITIONALITY *UNLESS_PAR_i*

$$({}_P\vdash p \text{ unless } q) \ \wedge \ ({}_Q\vdash p \text{ unless } q) \quad = \quad ({}_{P[]Q}\vdash p \text{ unless } q)$$

Corollary 4.7.2 $\circlearrowleft$ COMPOSITIONALITY *STABLE_PAR_i*

$$({}_P\vdash \circlearrowleft J) \ \wedge \ ({}_Q\vdash \circlearrowleft J) \quad = \quad ({}_{P[]Q}\vdash \circlearrowleft J)$$

Follows from unless COMPOSITIONALITY and the definition of $\circlearrowleft$.

Corollary 4.7.3 $\boxed{\mathsf{S}}$ COMPOSITIONALITY *Inv_PAR*

$$\frac{({}_P\vdash \boxed{\mathsf{S}}\, J) \ \wedge \ ({}_Q\vdash \boxed{\mathsf{S}}\, J)}{{}_{P[]Q}\vdash \boxed{\mathsf{S}}\, J}$$

Follows from $\circlearrowleft$ COMPOSITIONALITY and the definition of $\boxed{\mathsf{S}}$.

Theorem 4.7.4 ensures COMPOSITIONALITY *ENSURES_PAR*

$$\frac{({}_P\vdash p \text{ ensures } q) \ \wedge \ ({}_Q\vdash p \text{ unless } q)}{{}_{P[]Q}\vdash p \text{ ensures } q}$$

◀

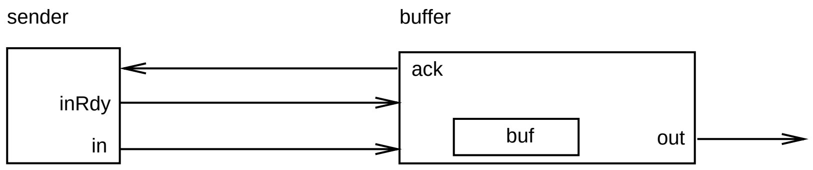
Figure 4.13: *Compositionality of safety properties and of* ensures.

In $\rightarrowtail$, the J is only required to be stable, which apparently is enough to have the SUBSTITUTION law. This is a useful generalization because when combining parallel programs, an invariant is easier to be destroyed than a stable predicate. Also, since $\overset{\mathsf{S}}{\mapsto}$ is based on $\mapsto$, which is not confined by $\mathbf{w}P$, it will have the same problem as $\mapsto$ when it comes to parallel composition, especially the parallel composition of write-disjoint programs. If one wishes to combine the strength of Sanders' definition and that of $\rightarrowtail$, one will have to parameterize with both stable predicates and invariants.

4.7 Parallel Composition

As has been motivated in Chapter 2, composition laws are useful as they enable us to decompose a global specification of a program into local specifications of the program's components. Not only that the original problem, which may contain complicated inter-component dependencies, is thereby broken into more manageable pieces, but also each component can subsequently be developed in isolation. In this section an overview of various parallel composition laws that we have verified will be given.

The compositionality of safety properties follows a very simple principle: the safety of a composite follows from the safety of its components; the laws are given in Figure 4.13. As for the compositionality of progress properties, only the compositionality of ensures was known in the first place. A parallel component may write to a common variable, and thereby affecting, or even, destroying a progress property of another com-



```
prog    Sender
read    {ack, in, inRdy}
write   {in, inRdy}
init    ¬inRdy
assign  if ¬inRdy ∧ ¬ack then in, inRdy := "generate a new datum", true
[]      if inRdy ∧ ack then inRdy := false

prog    Buffer
read    {in, inRdy, ack, buf, out}
write   {ack, buf, out}
init    ¬ack ∧ (buf = [])
assign  if inRdy ∧ ¬ack ∧ (ℓ.buf < N) then buf, ack  :=  buf ++[in], true
[]      if ¬inRdy ∧ ack then ack := false
[]      if buf ≠ [] then out, buf  :=  hd.buf, tl.buf
```

Figure 4.14: *A sender and an N-placed buffer.*

ponent. The phenomenon was not well understood and it was thought that except in the most trivial cases, no useful result can be obtained for progress properties expressed by $\mapsto$. A step forward is made by Singh [Sin89]. Although no claim is made on the strength of Singh's results, we believe that it is fairly strong. The results will be discussed in the sequel. However, instead of $\mapsto$, $\rightarrowtail$ will be used to express progress. The laws will look slightly different but the idea remains the same.

Stronger results can be obtained for parallel composition of programs which share no common write variables, but $\rightarrowtail$ is really required here. Perhaps, it should also be noted that as we attempted to verify Singh's results it was discovered that his main theorem is flawed. The flaw may look trivial and can be easily removed, but a considerably more sophisticated proof is required. This should illustrate a bit as how poor the issue was —and probably still is— understood.

4.7.1 General Progress Composition

Consider again the program Buffer in Figure 4.10 and a new program Sender, both displayed in Figure 4.14. Both programs are intended to be put together in parallel. The picture in Figure 4.14 may be helpful. The program Sender generates data and sends it through in to the program Buffer. The latter puts the data in an N-placed buffer buf and meanwhile, it also passes the data to out, first in, first out. The reader may notice that the synchronization between Sender and Buffer is a hand-shake protocol

$$\begin{tikzcd}
\neg\mathsf{inRdy} \wedge \neg\mathsf{ack} \arrow[r, "\alpha", "S"'] & \mathsf{inRdy} \wedge \neg\mathsf{ack} \arrow[loop right, "N\leq\ell.\mathsf{buf}"] \arrow[d, "B"', "\beta"] \\
\neg\mathsf{inRdy} \wedge \mathsf{ack} \arrow[u, "B"] & \mathsf{inRdy} \wedge \mathsf{ack} \arrow[l, "S"]
\end{tikzcd}$$

Note: S abbreviates Sender and B Buffer. α produces a new datum and β put the datum in buf, if there is a place for it.

Figure 4.15: *A 4-phase hand-shake protocol between* Sender *and* Buffer.

with the following phases:

$$\begin{array}{lcrcr}
\text{phase } 0 & : & \neg\mathsf{inRdy} & \wedge & \neg\mathsf{ack} \\
\text{phase } 1 & : & \mathsf{inRdy} & \wedge & \neg\mathsf{ack} \\
\text{phase } 2 & : & \mathsf{inRdy} & \wedge & \mathsf{ack} \\
\text{phase } 3 & : & \neg\mathsf{inRdy} & \wedge & \mathsf{ack}
\end{array}$$

A new datum can only be generated in phase 0, which subsequently brings the system to phase 1. A new datum can only enter buf in phase 1 (and if buf has a place free), which subsequently brings the system to phase 2. Then, Sender will flip its inRdy, bringing the system into phase 3, and Buffer its ack, reverting the system to phase 0. The transition graph in Figure 4.15 may be helpful.

Let SB abbreviate Sender $[\!]$ Buffer. A fundamental property which we claim that SB has is the following:

$$(\forall X :: \mathsf{true}\ {}_{\mathsf{SB}}\!\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X)) \tag{4.7.1}$$

stating that a new, un-acknowledged datum will eventually reach out. This progress property can be proved directly from the code of SB. However, let us now see how it can be derived from the properties of Sender and Buffer. Let us first do some calculation, starting from (4.7.1):

$$\begin{array}{ll}
 & (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X) \\
\Leftarrow & \quad \{ \rightarrowtail \textsc{Transitivity}_{56} \} \\
 & ((\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail X \in \mathsf{buf}) \ \wedge \ (X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X))
\end{array}$$

Now, by applying $\rightarrowtail$ PSP_{56} to the first conjunct using the following instantiation of the law:

$$\begin{array}{lcl}
p & \leftarrow & (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \\
q & \leftarrow & X \in \mathsf{buf} \vee (\mathsf{in} \neq X) \vee \neg\mathsf{inRdy} \vee \mathsf{ack} \\
r & = & p \\
s & \leftarrow & X \in \mathsf{buf} \\
J & \leftarrow & \mathsf{true}
\end{array}$$

we can refine the last specification to the following:

$$\begin{array}{lll} & {}_{\mathsf{SB}}\vdash & (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack}\ \mathsf{unless}\ X \in \mathsf{buf} \quad (4.7.2) \\ \mathsf{true} & {}_{\mathsf{SB}}\vdash & (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail X \in \mathsf{buf} \vee (\mathsf{in} \neq X) \vee \neg\mathsf{inRdy} \vee \mathsf{ack} \quad (4.7.3) \\ \mathsf{true} & {}_{\mathsf{SB}}\vdash & X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X) \quad (4.7.4) \end{array}$$

(4.7.2) states that while in phase 1 the system (SB) can only either remain there, or put the value of in into buf. We leave it to the reader to figure it out why SB satisfies this. (4.7.4) should be provable from the fact that this is a 'local' progress of the Buffer. We will return to this later. Let us for now concentrate on (4.7.3). Something that might help to prove this is the following property of Buffer:

$$(\mathsf{in} = X) \wedge \mathsf{inRdy}\ {}_{\mathsf{Buffer}}\vdash\ \mathsf{true} \rightarrowtail X \in \mathsf{buf} \vee \mathsf{ack} \quad (4.7.5)$$

But how can this be exploited to prove (4.7.3)? Note that the only way Sender can influence Buffer is through the variables in and inRdy. However, Sender *cannot* modify any of those if $\mathsf{inRdy} \wedge \neg\mathsf{ack}$ holds. Consequently, if in addition $\mathsf{in} = X$ then either by (4.7.5) Buffer will make its progress to $X \in \mathsf{buf} \vee \mathsf{ack}$, or either program does something to inRdy, ack, or in which it invalidates either $\mathsf{inRdy} \wedge \neg\mathsf{ack}$ or $\mathsf{in} = X$. In other words, (4.7.3) is implied!

The principle used to conclude (4.7.3) from(4.7.5) is an instantiation of a composition law known as Singh's Law [Sin89]. Before it can be presented, first we need some definitions.

Definition 4.7.5 !? *DVa*

$$Q!?P \;=\; \mathbf{w}Q \cap \mathbf{r}P$$

◀

So, $Q!?P$ denotes the variables through which Q can influence P. For example, Sender!?Buffer is $\{\mathsf{in}, \mathsf{inRdy}\}$. In practice people often use the term 'shared variables'. This term is ambiguous because it is not clear whether it is meant variables which are read, or written in common, or some other combination.

Let V be a set of variables. Let ${}_{Q}\vdash p\ \mathsf{unless}_V\ q$ roughly mean that under condition p, Q cannot alter the variables in V without establishing q:

Definition 4.7.6 unless_V

$${}_{Q}\vdash p\ \mathsf{unless}_V\ q \;=\; (\forall X :: {}_{Q}\vdash p \wedge (\forall v : v \in V : v = X.v)\ \mathsf{unless}\ q)$$

◀

Note: the dummy X has the type Var→Val. By omitting some overloading we can also write the formula above as:

$${}_{Q}\vdash p\ \mathsf{unless}_V\ q \;=\; (\forall X :: {}_{Q}\vdash p \wedge (\lambda s.\ s \restriction V = X \restriction V)\ \mathsf{unless}\ q)$$

In particular, ${}_Q\vdash p\ \mathsf{unless}_{Q!?P}\ q$ means that under condition p, Q *cannot* influence P without establishing q. For example, ${}_Q\vdash p\ \mathsf{unless}_{Q!?P}\ \mathsf{false}$ means that Q cannot influence P as long as p holds; ${}_Q\vdash \mathsf{true}\ \mathsf{unless}_{Q!?P}\ q$ means that Q always marks its interference to P by establishing q. Recall that the program Sender cannot influence Buffer as long as $\mathsf{inRdy} \wedge \neg\mathsf{ack}$ holds. So, Sender satisfies:

$$_{\mathsf{Sender}}\vdash \mathsf{inRdy} \wedge \neg\mathsf{ack}\ \mathsf{unless}_{\mathsf{Sender!?Buffer}}\ \mathsf{false} \tag{4.7.6}$$

There are two lemmas that we are going to use later. The first states that if p is a predicate confined by $\mathbf{r}P$, then Q cannot destroy p without changing a variable in $Q!?P$:

Lemma 4.7.7 *CONF_SAFE*

$$\frac{p \in \mathsf{Pred}.(\mathbf{r}P)}{{}_Q\vdash p \wedge (\lambda s.\ s{\restriction}U = X{\restriction}U)\ \mathsf{unless}\ (\lambda s.\ s{\restriction}U \neq X{\restriction}U)} \quad \text{where } U = Q!?P$$

▶

To prove the above lemma, the theory developed in Chapter 3 will now be brought into play.

Proof:

$$\begin{array}{ll}
 & p \wedge (\lambda s.\ s{\restriction}U = X{\restriction}U)\ \mathsf{unless}\ (\lambda s.\ s{\restriction}U \neq X{\restriction}U) \\
= & \{\text{ definition } \mathsf{unless}\text{, predicate calculus }\} \\
 & (\forall a : a \in \mathbf{a}Q : \{p \wedge (\lambda s.\ s{\restriction}U = X{\restriction}U)\}\ a\ \{(\lambda s.\ s{\restriction}U = X{\restriction}U) \Rightarrow p\}) \\
\Leftarrow & \{\ U = \mathbf{w}Q \cap \mathbf{r}P\text{, Theorem } 3.4.5_{33}\ \} \\
 & p \in \mathsf{Pred}.(\mathbf{r}P)\ \wedge\ (\forall a : a \in \mathbf{a}Q : (\mathbf{w}Q)^{\mathsf{c}} \not\leftarrow a) \\
\Leftarrow & \{\text{ definition } \mathsf{Unity}\ \} \\
 & p \in \mathsf{Pred}.(\mathbf{r}P)\ \wedge\ \mathsf{Unity}.Q
\end{array}$$

▲

Lemma 4.7.8

$$\frac{p \in \mathsf{Pred}.(\mathbf{r}P)\ \wedge\ ({}_Q\vdash r\ \mathsf{unless}_{Q!?P}\ s)}{{}_Q\vdash p \wedge r\ \mathsf{unless}\ s}$$

▶

Proof:
Let $U = Q!?P$. We want to prove $p \wedge r\ \mathsf{unless}\ s$. Using unless SIMPLE DISJUNCTION$_{46}$ it suffices to show that for all X:

$$p \wedge r \wedge (\lambda s.\ s{\restriction}U = X{\restriction}U)\ \mathsf{unless}\ s$$

Using unless CONJUNCTION$_{46}$ and POST WEAKENING$_{46}$ it suffices to show:

$$\begin{array}{lll}
r \wedge (\lambda s.\ s{\restriction}U = X{\restriction}U) & \mathsf{unless} & s \\
p \wedge (\lambda s.\ s{\restriction}U = X{\restriction}U) & \mathsf{unless} & (\lambda s.\ s{\restriction}U \neq X{\restriction}U)
\end{array}$$

The first follows from ${}_Q\vdash r$ unless_U s. The second follows from Lemma 4.7.7.
▲

Now here is the Singh law. If P can make progress $p \rightarrowtail q$, and under condition r, Q cannot influence P without establishing s, then in the composition $P[\![Q$ starting from p and r, either P makes its progress to q, or it does something that invalidates r, or Q writes something to P, in which case s will hold. The law is formulated below. A more general version is also provided.

Theorem 4.7.9 Singh Law *REACH_SINGH*

$$\frac{r, s \in \mathsf{Pred}.\mathbf{w}(P[\![Q) \;\wedge\; ({}_Q\vdash \circlearrowleft J) \;\wedge\; ({}_Q\vdash J \wedge r \;\mathsf{unless}_{Q!?P}\; s) \;\wedge\; (J \;{}_P\vdash p \rightarrowtail q)}{J \;{}_{P[\![Q}\vdash p \wedge r \rightarrowtail q \vee \neg r \vee s}$$

Theorem 4.7.10 (General) Singh Law *REACH_SINGH_g*

$$\frac{\begin{array}{c} r, s \in \mathsf{Pred}.\mathbf{w}(P[\![Q) \;\wedge\; p_1 \in \mathsf{Pred}.(\mathbf{w}P \cup (Q!?P)) \\ ({}_{P[\![Q}\vdash \circlearrowleft J) \;\wedge\; ({}_Q\vdash J \wedge r \;\mathsf{unless}_{Q!?P}\; s) \;\wedge\; (J \wedge p_1 \;{}_P\vdash p_2 \rightarrowtail q) \end{array}}{J \;{}_{P[\![Q}\vdash p_1 \wedge p_2 \wedge r \rightarrowtail q \vee \neg p_1 \vee \neg r \vee s}$$

▶

Only the proof of the General Singh Law will be presented.
Proof:
By applying $\rightarrowtail$ Induction$_{56}$, it suffices to show that $\mathfrak{R} = (\lambda a, b.\; J \;{}_{P[\![Q}\vdash p_1 \wedge a \wedge r \rightarrowtail b \vee \neg p_1 \vee \neg r \vee s)$ is transitive, left-disjunctive, and includes $\mathfrak{E} = (\lambda a, b.\; J \wedge p_1 \;{}_P\vdash a \;\mathfrak{ensures}\; b)$. To show that $\mathfrak{R}$ is transitive is easy:

$$\begin{array}{ll} & (p_1 \wedge a \wedge r \rightarrowtail b \vee \neg p_1 \vee \neg r \vee s) \;\wedge\; (p_1 \wedge b \wedge r \rightarrowtail c \vee \neg p_1 \vee \neg r \vee s) \\ = & \{ \text{ predicate calculus } \} \\ & (p_1 \wedge a \wedge r \rightarrowtail (p_1 \wedge b \wedge \neg r) \vee \neg p_1 \vee r \vee s) \;\wedge\; (p_1 \wedge b \wedge r \rightarrowtail c \vee \neg p_1 \vee \neg r \vee s) \\ \Rightarrow & \{ \rightarrowtail \textsc{Cancellation}_{56} \} \\ & p_1 \wedge a \wedge r \rightarrowtail c \vee \neg p_1 \vee \neg r \vee s \end{array}$$

The left-disjunctivity of $\mathfrak{R}$ follows directly from $\rightarrowtail$ Disjunction$_{56}$. It remains to show that $\mathfrak{R}$ includes $\mathfrak{E}$.

By applying $\rightarrowtail$ Introduction$_{56}$ and $\mathsf{ensures}$ Compositionality$_{60}$, $\mathfrak{R}.a.b$ is implied by:

i. $(p_1 \wedge a \wedge r) \;\in\; \mathsf{Pred}.(\mathbf{w}P[\![Q)$

ii. $(b \vee \neg p_1 \vee \neg r \vee s) \;\in\; \mathsf{Pred}.(\mathbf{w}P[\![Q)$

iii. ${}_{P[\![Q}\vdash \;\circlearrowleft J$

iv. ${}_P\vdash\; J \wedge p_1 \wedge a \wedge r \;\mathsf{ensures}\; b \vee \neg p_1 \vee \neg r \vee s$

v. ${}_Q\vdash\; J \wedge p_1 \wedge a \wedge r \;\mathsf{unless}\; b \vee \neg p_1 \vee \neg r \vee s$

The first two are easy and left to the reader. ***iii*** appears in the assumption. Using $\mathsf{ensures}$ Post-weakening$_{47}$, ***iv*** is implied by ${}_P\vdash J \wedge p_1 \wedge a \wedge r$ $\mathsf{ensures}$ b, which follows from $\mathfrak{E}.a.b$. For ***v*** we derive:

$J \wedge p_1 \wedge a \wedge r$ **unless** $b \vee \neg p_1 \vee \neg r \vee s$

$\Leftarrow$ { **unless** Post-weakening$_{46}$ }

$J \wedge p_1 \wedge a \wedge r$ **unless** s

$\Leftarrow$ { ${}_{P[]Q}\vdash \circlearrowright J$ implies ${}_{Q}\vdash \circlearrowright J$, **unless** Simple Conjunction$_{46}$ }

$p_1 \wedge a \wedge r$ **unless** s

$\Leftarrow$ { Lemma 4.7.8 }

$(p_1 \wedge a) \in \mathsf{Pred}.(\mathbf{r}P) \ \wedge \ r \ \mathsf{unless}_{Q!?P} \ s$

$\Leftarrow$ { predicate confinement distributes over $\wedge$; assumptions }

$p_1 \in \mathsf{Pred}.(\mathbf{r}P) \ \wedge \ a \in \mathsf{Pred}.(\mathbf{r}P)$

$\Leftarrow$ { confinement is monotonic with respect to $\subseteq$; definition $\mathfrak{C}$ }

$p_1 \in \mathsf{Pred}.(\mathbf{w}P \cup (Q!?P)) \wedge \mathfrak{C}.a.b$

▲

Let us now try to apply the law to our example with Sender and Buffer. Recall that we want to obtain the specification (4.7.3) from (4.7.5). The first is re-displayed below:

$$\mathsf{true} \ {}_{\mathsf{SB}}\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail X \in \mathsf{buf} \vee (\mathsf{in} \neq X) \vee \neg\mathsf{inRdy} \vee \mathsf{ack}$$

By applying the General Singh Law, we can refine the above to the following:

i. $(\mathsf{inRdy} \wedge \neg\mathsf{ack}) \in \mathsf{Pred}.(\mathbf{w}(\mathsf{Sender}[]\mathsf{Buffer}))$

ii. $\mathsf{false} \in \mathsf{Pred}.(\mathbf{w}(\mathsf{Sender}[]\mathsf{Buffer}))$

iii. $(\mathsf{in} = X) \wedge \mathsf{inRdy}) \in \mathsf{Pred}.(\mathbf{w}(\mathsf{Buffer}) \cup \mathsf{Sender}!?\mathsf{Buffer})$

iv. $(\mathsf{in} = X) \wedge \mathsf{inRdy} \ {}_{\mathsf{Buffer}}\vdash \ \mathsf{true} \rightarrowtail X \in \mathsf{buf} \vee \mathsf{ack}$

v. ${}_{\mathsf{Sender}}\vdash \ \mathsf{inRdy} \wedge \neg\mathsf{ack} \ \mathsf{unless}_{\mathsf{Sender}!?\mathsf{Buffer}} \ \mathsf{false}$

The first three are trivial. ***iv*** is (4.7.5). ***v*** is (4.7.6), stating that Sender cannot influence Buffer as long as inRdy and ¬ack hold. It is not too difficult to conclude from its code that Sender has this property.

The Singh Law, although it formulates a very intuitive idea, looks complicated. The law reduces a progress specification of a program to a progress specification of one of its component, two safety specifications, and some 'type' restrictions on the predicates that occur in the specifications. At first sight, applying the law seems only to generate more proof obligations and one may therefore question the merit of using the law. However, recall that a $\rightarrowtail$ (or $\mapsto$) progress property is proved by composing a number of **ensures** properties. Without the Singh Law, or some other parallel composition law, all these **ensures** properties will have to be verified with respect to the whole program. With the Singh Law they only have to be verified with respect to a component program. If the number of **ensures** properties and the average size of the component programs are sufficiently large, then applying the Singh Law will become more economical.

In some cases, the law can be simplified. The following corollaries show some of these cases. In the linear temporal logic there is a relation called **until**. A program P satisfies ${}_{P}\vdash p$ **until** q if whenever p holds during an execution of P, afterwards q will eventually hold but in the meantime p will continue to hold until q holds. This sounds very much like $\mathsf{ensures}$, but until is actually larger than $\mathsf{ensures}$. Roughly speaking, until is equal to $\mapsto \cap$ unless . Consequently, either $p\ \vdash \mathsf{true} \rightarrowtail q$ or $(\mathsf{true}\ \vdash p \rightarrowtail q) \wedge (\ \vdash p\ \mathsf{unless}\ q)$ imply $p\ \mathsf{until}\ q$. If either holds in P, and under condition p, Q cannot influence P without establishing q then one can conclude that $p \rightarrowtail q$ must hold in the composition $P\|Q$. This is what the following two corollaries state.

Corollary 4.7.11 until COMPOSITIONALITY *UNTIL_COMPO1*

$$\frac{({}_{Q}\!\vdash \circlearrowleft J) \ \wedge\ ({}_{Q}\!\vdash J \wedge p\ \mathsf{unless}_{P?!Q}\ q) \ \wedge\ ({}_{P}\!\vdash J \wedge p\ \mathsf{unless}\ q) \ \wedge\ (J\ {}_{P}\!\vdash p \rightarrowtail q)}{J\ {}_{P\|Q}\!\vdash p \rightarrowtail q}$$

Corollary 4.7.12 until COMPOSITIONALITY *UNTIL_COMPO2*

$$\frac{\begin{array}{c} p \in \mathsf{Pred}.(\mathbf{w}P \cup (P?!Q)) \ \wedge\ ({}_{P\|Q}\!\vdash \circlearrowleft J) \\ ({}_{Q}\!\vdash J \wedge p\ \mathsf{unless}_{P?!Q}\ q) \ \wedge\ (J \wedge p\ {}_{P}\!\vdash \mathsf{true} \rightarrowtail q) \end{array}}{J\ {}_{P\|Q}\!\vdash p \rightarrowtail q}$$

◀

4.7.2 Exploiting Fix Point

A state s is called a fix point of a program P, if s remains unchanged under the execution of any action in P. The notion can be lifted to the predicate level. A predicate p is called a fix point of P, denoted by $p \in \mathsf{Fp}.P$, if all states $s \in p$ are fix points.

Definition 4.7.13 FIX POINT *FPp_DEF*

$$p \in \mathsf{Fp}.P \ = \ (\forall a, s : a \in \mathbf{a}P : p.s \wedge a.s.t \Rightarrow (s = t))$$

◀

Note that the definition above may not match its intended interpretation if P contains an action a which is not always-enabled (that is, for some begin state s, a may not be able to make any transition). However, in the case of UNITY programs, their actions are assumed to be always-enabled.

Fix points can be useful in parallel composition. If a program P has reached a fix point, then certainly it cannot influence any other program Q. If Q cannot throw P from its fix points space, then any progress in Q will be preserved in the composition $P\|Q$. In fact, this principle is a corollary of the Singh law. Before its formulation is presented here are some basic properties of fix points. Fix points are anti-monotonic with respect to $\Rightarrow$. In addition, if a predicate q is a fix-point of P, then for any p, $p \wedge q$ is stable in P.

Theorem 4.7.14 Fp MONOTONICITY *FPp_MONO*

$$\frac{[p \Rightarrow q] \;\wedge\; q \in \mathsf{Fp}.P}{p \in \mathsf{Fp}.P}$$

Theorem 4.7.15 Fp STABILITY *FPp_IMP_STABLE*

$$\frac{q \in \mathsf{Fp}.P}{(p \wedge q) \in \mathsf{Fp}.P}$$

◀

We now present the composition law using fix points we mentioned before[8]:

Theorem 4.7.16 *REACH_COMPO_BY_FPp*

$$\frac{({}_{P}\vdash \circlearrowright(J_1 \wedge J_2)) \;\wedge\; J_2 \in \mathsf{Fp}.Q \;\wedge\; (J_1 \;{}_{P}\vdash p \rightarrowtail q)}{J_1 \wedge J_2 \;{}_{P[\!]Q}\vdash p \rightarrowtail q}$$

▶

Proof:

$$\begin{array}{ll}
& J_2 \in \mathsf{Fp}.Q \;\wedge\; (J_1 \;{}_{P}\vdash p \rightarrowtail q) \\
\Rightarrow & \{ \text{ Theorem 4.7.15; definition of } \mathsf{unless}_V \} \\
& (\,{}_{Q}\vdash J_1 \wedge J_2 \wedge \mathsf{true\ unless}_{Q!?P}\ \mathsf{false}) \;\wedge\; (J_1 \;{}_{P}\vdash p \rightarrowtail q) \\
\Rightarrow & \{ \;{}_{P}\vdash \circlearrowright(J_1 \wedge J_2),\; \circlearrowright \text{CONJUNCTION}_{47} \} \\
& (\,{}_{Q}\vdash J_1 \wedge J_2 \wedge \mathsf{true\ unless}_{Q!?P}\ \mathsf{false}) \;\wedge\; (J_1 \wedge J_2 \;{}_{P}\vdash p \rightarrowtail q) \\
\Rightarrow & \{ \text{ SINGH LAW}_{65} \} \\
& J_1 \wedge J_2 \;{}_{P[\!]Q}\vdash p \rightarrowtail q
\end{array}$$

▲

4.7.3 Why the Original Version of Singh Law is Flawed

Earlier it was mentioned that the original version of the Singh law is flawed. Let us now take a look at this and see what we can learn from it. However, if the reader wishes, he can skip this entire subsection.

A simple form of the original Singh Law [Sin89] looks as follow:

$$\frac{(\,{}_{Q}\vdash \mathsf{true\ unless}_{Q!?P}\ s) \;\wedge\; (\,{}_{P}\vdash p \mapsto q)}{{}_{P[\!]Q}\vdash p \mapsto q \vee s} \qquad (4.7.7)$$

stating that if Q cannot influence P without establishing s, then any progress made by P will be preserved in the composition $P[\!]Q$, or Q interferes and establishes s.

Note that *there is no restriction on p, q and s.* They may be any predicates! Now consider the following programs:

[8] The law was given first by Singh in [Sin89].

prog	P	prog	Q
read	$\{x\}$	read	$\{a\}$
write	$\{x\}$	write	$\{a\}$
init	true	init	true
assign	$x := x + 1$	assign	$a := \neg a$

Since the programs share no variable then obviously Q cannot influence P ($Q!?P = \emptyset$), and hence it satisfies:

$$_Q\vdash \textsf{true unless}_{Q!?P}\ \textsf{false} \tag{4.7.8}$$

A valid property of P is $_P\vdash a \wedge (x = 0) \mapsto a \wedge (x = 1)$. Using (4.7.8) and Singh law (4.7.7) we conclude the progress also holds in $P[\!]Q$. But this cannot of course be true.

As no restriction is put on p and q in (4.7.7), the progress $p \mapsto q$ may actually refer to some internal variable of Q, and this is what causes the problem above. Let us now put a restriction on them and see how we can prove the law:

$$\frac{p, q \in \textsf{Pred.}(\mathbf{r}P)\ \wedge\ (_Q\vdash \textsf{true unless}_{Q!?P}\ s)\ \wedge\ (_P\vdash p \mapsto q)}{_{P[\!]Q}\vdash p \mapsto q \vee s} \tag{4.7.9}$$

To prove the above we do not have many options but to resort to $\mapsto$ INDUCTION$_{52}$. So, assuming $_Q\vdash \textsf{true unless}_{Q!?P}\ r$, prove that $\mathfrak{R} = (\lambda p, q.\ p, q \in \textsf{Pred.}(\mathbf{r}P) \Rightarrow (\ _{P[\!]Q}\vdash p \mapsto q \vee s))$ is transitive, left-disjunctive, and includes **ensures**. But we have now a new problem. The transitivity (and left-disjunctivity) of $\mathfrak{R}$ cannot be proved:

$$p, q \in \textsf{Pred.}(\mathbf{r}P) \Rightarrow (\ _{P[\!]Q}\vdash p \mapsto q \vee s))$$
$$q, r \in \textsf{Pred.}(\mathbf{r}P) \Rightarrow (\ _{P[\!]Q}\vdash q \mapsto r \vee s))$$

are not sufficient to prove

$$p, r \in \textsf{Pred.}(\mathbf{r}P) \Rightarrow (\ _{P[\!]Q}\vdash p \mapsto r \vee s))$$

So, let us instead prove a slightly different law:

Theorem 4.7.17 SIMPLE SINGH LAW FOR $\mapsto$

$$\frac{(\ _Q\vdash \textsf{true unless}_{Q!?P}\ s)\ \wedge\ (\ _P\vdash p \mapsto q)}{_{P[\!]Q}\vdash (p \upharpoonright \mathbf{r}P) \mapsto (q \upharpoonright \mathbf{r}P) \vee s}$$

Note that if $p, q \in \textsf{Pred.}(\mathbf{r}P)$, hence in other words $(p = p \upharpoonright \mathbf{r}P)\ \wedge\ (q = q \upharpoonright \mathbf{r}P)$, the above implies (4.7.9).

Proof:

Assuming $_Q\vdash \textsf{true unless}_{Q!?P}\ s$, the law above will be proved using $\mapsto$ INDUCTION$_{52}$. So, it will be showed that $\mathfrak{R} = (\lambda p, q.\ _{P[\!]Q}\vdash (p \upharpoonright \mathbf{r}P) \mapsto (q \upharpoonright \mathbf{r}P) \vee s)$ is transitive, left-disjunctive, and includes $(\lambda p, q.\ _P\vdash p$ **ensures** $q)$. For the transitivity:

$$((p \upharpoonright \mathbf{r}P) \mapsto (q \upharpoonright \mathbf{r}P) \vee s)\ \wedge\ ((q \upharpoonright \mathbf{r}P) \mapsto (r \upharpoonright \mathbf{r}P) \vee s)$$

$\Rightarrow \quad \{ \mapsto \textsc{Cancellation}_{52} \}$

$\quad (p \restriction \mathbf{r}P) \mapsto (r \restriction \mathbf{r}P) \vee s$

The left-disjunctivity of $\mathfrak{R}$ follows directly from $\mapsto \textsc{Disjunction}_{52}$ and the fact that the confinement of predicates distributes over $\vee$.

As for the inclusion of $\mathsf{ensures}$, assume $_P\vdash p \;\mathsf{ensures}\; q$. We derive:

$\quad {}_{P[]Q}\vdash (p \restriction \mathbf{r}P) \mapsto (q \restriction \mathbf{r}P) \vee s$

$\Leftarrow \quad \{ \mapsto \textsc{Introduction}_{52} \}$

$\quad {}_{P[]Q}\vdash (p \restriction \mathbf{r}P) \;\mathsf{ensures}\; (q \restriction \mathbf{r}P) \vee s$

$\Leftarrow \quad \{ \mathsf{ensures}\; \textsc{Compositionality}_{60} \}$

$\quad ({}_P\vdash (p \restriction \mathbf{r}P) \;\mathsf{ensures}\; (q \restriction \mathbf{r}P) \vee s) \;\wedge\; ({}_Q\vdash (p \restriction \mathbf{r}P) \;\mathsf{unless}\; (q \restriction \mathbf{r}P) \vee s)$

$\Leftarrow \quad \{ \mathsf{ensures}\; \textsc{Post-weakening}_{47};\ \text{Corollary } 3.4.3_{31} \}$

$\quad (\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\rightarrow a) \;\wedge\; ({}_P\vdash p \;\mathsf{ensures}\; q) \;\wedge\; ({}_Q\vdash (p \restriction \mathbf{r}P) \;\mathsf{unless}\; (q \restriction \mathbf{r}P) \vee s)$

$\Leftarrow \quad \{ \text{assumption, definition } \mathsf{Unity}_{40} \}$

$\quad \mathsf{Unity}.P \;\wedge\; ({}_Q\vdash (p \restriction \mathbf{r}P) \;\mathsf{unless}\; (q \restriction \mathbf{r}P) \vee s)$

$\Leftarrow \quad \{ \mathsf{unless}\; \textsc{Post-weakening}_{46} \text{ and Lemma } 4.7.8_{64} \}$

$\quad \mathsf{Unity}.P \;\wedge\; p \restriction \mathbf{r}P \in \mathsf{Pred}.(\mathbf{r}P) \;\wedge\; ({}_Q\vdash \mathsf{true}\; \mathsf{unless}_{Q!?P}\; s)$

$= \quad \{ P \text{ is a UNITY program, } p \restriction V \text{ is always confined by } \mathsf{Pred}.V \}$

$\quad {}_Q\vdash \mathsf{true}\; \mathsf{unless}_{Q!?P}\; s$

▲

4.7.4 A Short Overview

At this point, the reader may begin to lose track as to where we are aiming at. Just to remind him: we have introduced the logic UNITY and discussed how to represent a UNITY program. We have presented the standard UNITY operators, used to express the behaviors of a program, and discussed their shortcomings. We gave a special attention to progress properties and extended UNITY with a new progress operator which we claim to have a nice compositional property —this will be made clear in the next section. In the mean time, many calculational laws concerning the extended logic have been presented. Some of the laws will be used later, but some will not. In general though, the reader will likely find those laws to be useful for designing his own programs.

As said, the next section will present more compositionality results of the new progress operator $\rightarrowtail$. It is an important section. The two sections that follow the next section are included for the completeness sake. The first will discuss program transformations, which can be used as an alternative method to design a program as in [Bac90, R.95]. The second will present a standard operational semantics for UNITY and present some soundness results of the UNITY logic with respect to the mentioned semantics.

4.8 Write Disjoint Composition

Stronger compositionality results can be obtained for programs that are write-disjoint. Recall (from Chapter 3) that two programs P and Q are said to be write-disjoint if they write to no common variable. This is denoted by $P \div Q$. If two programs P and Q are write-disjoint, Q can only influence P through P's input variables, that is, the variables read by P but not written by it. Consequently, once P and Q agree on a set of input values for P, whatever progress P makes through its write variables will be preserved in the parallel composition of P and Q. In Chapter 3 we hypothesized a law called Transparency law which states exactly this. This will be proven in this section. The Transparency law is fundamental for write-disjoint composition. Some well known design techniques that we use in practice are corollaries of this principle. A progress property is usually constructed, either using transitivity or disjunction, from a number of simpler progress properties. Using the principle we can delegate each constituent property, if we so desire, to be realized by a write-disjoint component of a program.

Parallel composition of write-disjoint programs is also attractive because it occurs frequently in practice. The pictures in Figure 2.3 show various instances of composition of write-disjoint programs. We will define some of them below.

Definition 4.8.1 WRITE-DISJOINT PROGRAMS *WD_DEF*

$$P \div Q \;=\; (\mathbf{w}P \cap \mathbf{w}Q = \emptyset)$$

Definition 4.8.2 LAYERING *LAYERING*

$$P \triangleright Q \;=\; (P \div Q) \wedge (\mathbf{w}P \supseteq \mathbf{i}Q)$$

Definition 4.8.3 FORK *FORK*

$$P \pitchfork Q \;=\; (P \div Q) \wedge (\mathbf{i}P = \mathbf{i}Q)$$

Definition 4.8.4 NON-INTERFERING (PARALLEL) *FPAR*

$$P \| Q \;=\; (P \div Q) \wedge (\mathbf{r}P \cap \mathbf{r}Q = \emptyset)$$

◀

If $P \div Q$, then the parallel composition of P and Q is also called the *write-disjoint composition* of P and Q. Obviously, if $\triangleright$, $\pitchfork$, and $\|$ are all instances of $\div$.

In a *non-interfering parallel composition* of two programs, both programs are independent from each other. In a *fork*, the programs based their computation on the same set of input variables. For example if we have a program that computes the minimum of the values of the variables in V, and another program that computes the maximum, we can put the two programs in parallel by forking them. In a *layering* we have two layers. If $P \triangleright Q$ holds, then P is called the *lower layer* and Q the *upper layer*. The computation of the upper layer depends on the results of the lower layer.

The converse does not necessarily hold. For example, the lower layer can be a program that constructs a spanning tree from a vertex i and the upper layer is a program that broadcasts messages from i, using the constructed spanning tree. Layering works like a higher level sequential composition. However, the two layers do not have to be implemented sequentially, especially if they are non-terminating programs.

Before the Transparency law can be proven, first we need the following lemma, stating that if P and Q are write disjoint, then Q cannot destroy a predicate confined by $\mathbf{w}P$:

Lemma 4.8.5 *CONF_Local*

$$\frac{(P \div Q) \ \wedge \ p \in \mathsf{Pred}.(\mathbf{w}P)}{{}_Q\!\vdash \circlearrowleft p}$$

Proof:
The lemma follows from Lemma 3.4.6$_{33}$ in Chapter 3:

$$\begin{array}{ll}
 & {}_Q\!\vdash \circlearrowleft p \\
= & \{ \text{ definition of } \circlearrowleft (4.3.8) \ \} \\
 & (\forall a : a \in \mathbf{a}Q : \{p\} \ a \ \{p\}) \\
\Leftarrow & \{ \text{ Lemma } 3.4.6_{33} \ \} \\
 & (\forall a : a \in \mathbf{a}Q : \mathbf{w}P \not\leftarrow a \ \wedge \ p \in \mathsf{Pred}.(\mathbf{w}P)) \\
\Leftarrow & \{ \ P \text{ and } Q \text{ are write-disjoint } \} \\
 & p \in \mathsf{Pred}.(\mathbf{w}P)
\end{array}$$

▲

Now the transparency law:

Theorem 4.8.6 TRANSPARENCY LAW *REACH_TRANSPARANT*

$$\frac{P \div Q \ \wedge \ ({}_Q\!\vdash \circlearrowleft J) \ \wedge \ (J \ {}_P\!\vdash p \rightarrowtail q)}{J \ {}_{P[\!]Q}\!\vdash p \rightarrowtail q}$$

Proof:
By applying $\rightarrowtail$ INDUCTION$_{56}$ it suffices to show that $\mathfrak{R} = (\lambda p,q.\ J \ {}_{P[\!]Q}\!\vdash \ p \rightarrowtail q)$ is transitive, left-disjunctive, and includes $\mathfrak{E} = (\lambda p,q.\ J \ {}_P\!\vdash \ p \ \mathfrak{ensures} \ q)$. That $\mathfrak{R}$ is transitive and left-disjunctive follows from $\rightarrowtail$ TRANSITIVITY$_{56}$ and DISJUNCTIVITY$_{56}$. For the inclusion of $\mathfrak{E}$ we derive:

$$\begin{array}{ll}
 & J \ {}_{P[\!]Q}\!\vdash p \rightarrowtail q \\
\Leftarrow & \{ \rightarrowtail \text{INTRODUCTION}_{56} \ \} \\
 & p,q \in \mathsf{Pred}.(\mathbf{w}(P[\!]Q)) \ \wedge \ ({}_{P[\!]Q}\!\vdash \circlearrowleft J) \ \wedge \ ({}_{P[\!]Q}\!\vdash J \wedge p \ \mathsf{ensures} \ q) \\
\Leftarrow & \{ \ \mathbf{w}P \subseteq \mathbf{w}P[\!]Q; \text{ CONFINEMENT MONOTONICITY}_{30} \ \}
\end{array}$$

$$
\begin{array}{ll}
 & p, q \in \mathsf{Pred}.(\mathbf{w}P) \;\wedge\; ({}_{P\|Q}\vdash \circlearrowright J) \;\wedge\; ({}_{P\|Q}\vdash p \wedge J \;\mathsf{ensures}\; q) \\
= & \{\; \circlearrowright \textsc{Compositionality}_{60} \;\} \\
 & p, q \in \mathsf{Pred}.(\mathbf{w}P) \;\wedge\; ({}_{P}\vdash \circlearrowright J) \;\wedge\; ({}_{Q}\vdash \circlearrowright J) \;\wedge\; ({}_{P\|Q}\vdash p \wedge J \;\mathsf{ensures}\; q) \\
\Leftarrow & \{\; \mathsf{ensures}\ \textsc{Compositionality}_{60} \text{ and the definition}_{54} \text{ of } \mathfrak{ensures} \;\} \\
 & ({}_{Q}\vdash \circlearrowright J) \;\wedge\; ({}_{Q}\vdash p \wedge J \;\mathsf{unless}\; q) \;\wedge\; (J\ {}_{P}\vdash p \;\mathfrak{ensures}\; q) \\
\Leftarrow & \{\; \mathsf{unless}\ \textsc{Post-weakening}_{46}\text{; definition of } \circlearrowright\text{; } \circlearrowright \textsc{Conjunction}_{47} \;\} \\
 & ({}_{Q}\vdash \circlearrowright J) \;\wedge\; ({}_{Q}\vdash \circlearrowright p) \;\wedge\; (J\ {}_{P}\vdash p \;\mathfrak{ensures}\; q) \\
\Leftarrow & \{\; \text{Lemma 4.8.5} \;\} \\
 & ({}_{Q}\vdash \circlearrowright J) \;\wedge\; p \in \mathsf{Pred}.(\mathbf{w}P) \;\wedge\; (P \div Q) \;\wedge\; (J\ {}_{P}\vdash p \;\mathfrak{ensures}\; q) \\
= & \{\; \text{definition}_{54} \text{ of } \mathfrak{ensures} \;\} \\
 & ({}_{Q}\vdash \circlearrowright J) \;\wedge\; (P \div Q) \;\wedge\; (J\ {}_{P}\vdash p \;\mathfrak{ensures}\; q)
\end{array}
$$

▲

For example consider again the example with Sender and Buffer in Figure 4.14$_{61}$. Recall that we were discussing about the specification (4.7.1) of Sender‖Buffer:

$$(\forall X :: \mathsf{true}\ {}_{\mathsf{SB}}\vdash (\mathsf{in} = X) \wedge \mathsf{inRdy} \wedge \neg\mathsf{ack} \rightarrowtail (\mathsf{out} = X))$$

The specification was refined into a number of specifications. One of them is (4.7.4):

$$\mathsf{true}\ {}_{\mathsf{Sender\|Buffer}}\vdash X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X)$$

By looking into the code of Buffer we conclude that this progress will be established by Buffer, no matter what Sender does. But how can we conclude this without having to look into the code of Buffer? Notice that Sender and Buffer are write-disjoint. Using the Transparency law we can refine the above to:

$$\mathsf{true}\ {}_{\mathsf{Buffer}}\vdash X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X) \tag{4.8.1}$$

stating that the required progress can indeed be delegated to Buffer.

One may ask, whether a special law for write-disjoint programs is really necessary. That is, whether it can derived from the Singh Law given in the previous section. The latter expresses how, in general, a progress property of a program P may be influenced by another program Q (through the variables in $Q!?P$). However, the Singh Law does not discriminate between progress properties which are 'directly' dependent on $Q!?P$ and those that are not. For example, we cannot obtain (4.8.1) above from

$$\mathsf{true}\ {}_{\mathsf{Sender\|Buffer}}\vdash X \in \mathsf{buf} \rightarrowtail (\mathsf{out} = X)$$

using the Singh$_{65}$ law because the law *presumes* the worst case, which is that any progress made by Buffer may suffer from interference by Sender, which is not always true.

An instance of write-disjoint composition called *layering* —also called collateral composition— has been recognized by Herman [Her91] and Arora [Aro92] as an important technique to combine self-stabilizing programs. The following law is especially useful to handle layering.

Theorem 4.8.7 SPIRAL LAW *REACH_SPIRAL*

$$\frac{\begin{array}{c}(P \div Q) \ \wedge \ ({}_P\!\vdash \circlearrowright(J \wedge q)) \ \wedge \ ({}_Q\!\vdash \circlearrowright J) \\ (J \ {}_P\!\vdash p \rightarrowtail q) \ \wedge \ (J \wedge q \ {}_Q\!\vdash \mathsf{true} \rightarrowtail r)\end{array}}{J \ {}_{P[\![Q}\!\vdash p \rightarrowtail q \wedge r}$$

◀

If $P \triangleright Q$ holds, $P[\![Q$ is a layering with P as the lower layer and Q as the upper layer. According to the law, a progress property $p \rightarrowtail r$ in $P[\![Q$ can be split into $p \rightarrowtail q$ in the lower layer P, and $q \vdash \mathsf{true} \rightarrowtail r$ in the upper layer Q. The SPIRAL law is used to implement a sequential division of tasks. For example if we want to do a broadcast, we can think of a two-steps process: first, construct a spanning tree, and then do the actual broadcast. Usually we have separate programs for both tasks. The SPIRAL Law provides the required justification for this kind of separation, where in this case P constructs the spanning tree and Q performs the broadcast under the assumption that q describes the existence of this spanning tree.

4.9 Program Transformations

Parallel composition or sequential composition are, as the name implies, program composition in which two programs are combined to form a larger one. There are also program transformations in which a program is transformed into another one. In [CM88] only the addition of assignments to fresh variables is mentioned. There are of course more useful transformations but at the time not much was known about how exactly they affect the behavior of a program. Recent results were given by Singh in [Sin93] who exposed data refinement, guard strengthening, and refinement of atomicity, and investigated the kind of program properties preserved by these transformations. There is also the work by Udink, Herman, and Kok in [UHK94] which presented action duplication, data refinement, guard strengthening, and action substitution using invariants, and proved that these transformations preserved some form of local safety and progress properties.

In this section several program transformations will be discussed. The main question that we wish to address is how the transformations affect the $\rightarrowtail$ properties of a program. We expect that the results in [CM88] and [Sin93] for $\mapsto$ will translate well to results in $\rightarrowtail$. In addition, we find the laws of superposition (additions of assignments to fresh variables) in [CM88] to be somewhat informally stated. We will re-state them, with proper details. The reader may also find it interesting to see how the transformation laws presented later can be neatly proven from the laws at the action level presented in Chapter 3. The proofs are collected separately in Section 4.12. Program transformation is however not a main issue in this thesis. The technique is not going to be used in the applications presented later in this thesis. Therefore we are not going to be too elaborate. The results are also not mechanically verified yet.

If P is a UNITY program, adding variables to P, or a **skip** action, preserves the properness of P. That is, the resulting program also satisfies $\mathsf{Unity}.P$. Obviously, this simple transformation preserves whatever **unless**, **ensures**, $\rightarrowtail$, and invariant properties of P. In addition, strengthening the initial condition of a program also preserves such properties.

Recall than an action a can be extended with an assignment b to some fresh variables by composing b 'in front of' a: $b; a$. Assume that V are the variables read by a. Naturally, we expect that extending a with b preserves any Hoare triple specification of a, as long as the specification does not refer to the fresh variables. This is justified by Corollary 3.4.4$_{32}$ from Chapter 3 which states that $\{p\}\ a\ \{q\}$ implies $\{p \restriction V\}\ b; a\ \{q \restriction V\}$. The same should also hold at the program level. In UNITY, the addition of assignments to fresh variables is called *superposition*. In addition, if we can extend the actions in a program with assignments to fresh variables, we can also add new actions which only assign to fresh variables.

Theorem 4.9.1 PRIMITIVE PROPERTIES AFTER SUPERPOSITION
Let P and Q be UNITY programs and $f \in \mathsf{Action} \to \mathsf{Action}$ such that: $\mathbf{a}Q = \{f.a; a \mid a \in \mathbf{a}P\} \cup \{f.a \mid a \in A\}$ for some $A \subseteq \mathsf{Action}$. Let $\mathfrak{R}$ be either unless or $\mathsf{ensures}$. We have:

$$\frac{(\forall a : a \in \mathbf{a}P : \mathbf{r}P \not\leftarrow f.a) \ \wedge\ ({}_P\vdash p \,\mathfrak{R}\, q)}{{}_Q\vdash (p \restriction \mathbf{r}P) \,\mathfrak{R}\, (q \restriction \mathbf{r}P)}$$

Notice that Q is obtained from P by extending each action a of P with an assignment $f.a$ (which can also be skip). The first conjunct in the assumption of the above law expresses the fact that $f.a$ only assigns to fresh variables. A similar law also exists for $\rightarrowtail$. It can easily be proven using $\rightarrowtail$ INDUCTION$_{56}$.

Theorem 4.9.2 $\rightarrowtail$ AFTER SUPERPOSITION
Let P and Q be UNITY programs and $f \in \mathsf{Action}{\to}\mathsf{Action}$ such that $\mathbf{w}P \subseteq \mathbf{w}Q$ and: $\mathbf{a}Q = \{f.a; a \mid a \in \mathbf{a}P\} \cup \{f.a \mid a \in A\}$ for some $A \subseteq \mathsf{Action}$. We have:

$$\frac{(\forall a : a \in \mathbf{a}P : \mathbf{r}P \not\leftarrow f.a) \ \wedge\ (J\ {}_P\vdash p \rightarrowtail q)}{J \restriction \mathbf{r}P\ {}_Q\vdash p \rightarrowtail q}$$

An action a can be strengthened with a guard g by extending it to $\mathsf{if}\ g\ \mathsf{then}\ a$. The meaning of this action, according to the convention made Section 4.2 is:

$$(\lambda s, t.\ (g.s \Rightarrow a.s.t)\ \wedge\ (\neg g.s \Rightarrow (s = t)))$$

It is known that strengthening the actions of a program with guards preserves its safety properties:

Theorem 4.9.3 Safety under a Stronger Guard
Let P and Q be UNITY programs, $g \in \mathsf{Action} \to \mathsf{Pred}$, and $A \subseteq \mathbf{a}P$ such that $\mathbf{a}Q = (\mathbf{a}P \backslash A) \cup \{\mathsf{if}\ g.a\ \mathsf{then}\ a \mid a \in A\}$ We have:

$$\frac{{}_P\vdash p\ \mathsf{unless}\ q}{{}_Q\vdash p\ \mathsf{unless}\ q}$$

◀

Notice that Q is obtained from P by adding a guard $g.a$ to each action a from A.

If an action a in P is strengthened with a guard g, whatever progress by $\rightarrowtail$ in P will be preserved in the new program, *if* the other actions cannot destroy g, and if it will eventually hold. This is expressed by the following theorem[9].

Theorem 4.9.4 Progress under a Stronger Guard
Let P and Q be UNITY programs, $g \in \mathsf{Action} \to \mathsf{Pred}$, and $A \subseteq \mathbf{a}P$ such that $\mathbf{w}P \subseteq \mathbf{w}Q$ and $\mathbf{a}Q = (\mathbf{a}P \backslash A)\ \cup\ \{\mathsf{if}\ g.a\ \mathsf{then}\ a \mid a \in A\}$. Let Q_{-a} be the same program as Q, except that the action a is deleted. We have:

$$\frac{(\forall a : a \in A : ({}_{Q_{-a}}\vdash \circlearrowleft J \wedge g.a)\ \wedge\ (J\ {}_Q\vdash \mathsf{true} \rightarrowtail g.a))\ \wedge\ (J\ {}_P\vdash p \rightarrowtail q)}{J\ {}_Q\vdash p \rightarrowtail q}$$

◀

Notice that the condition $J\ {}_Q\vdash \mathsf{true} \rightarrowtail g.a$ states that in the new program Q, eventually the guard $g.a$ becomes true. The condition ${}_{Q_{-a}}\vdash \circlearrowleft J \wedge g$ states that no other action but a can falsify the guard $g.a$.

If ${}_P\vdash J \wedge p\ \mathsf{unless}\ q$ holds, and in addition J implies that $g = h$, then replacing an action $\mathsf{if}\ g\ \mathsf{then}\ a$ in P with $\mathsf{if}\ h\ \mathsf{then}\ a$ preserves $J \wedge p\ \mathsf{unless}\ q$. Typically J is an invariant, or at least a stable predicate. A similar result also exists for $\mathsf{ensures}$ and $\rightarrowtail$.

Theorem 4.9.5 Actions Substitution
Let P, Q and R be UNITY programs such that

$$Q = P [\![(\{\mathsf{if}\ g_1\ \mathsf{then}\ \mathsf{assign}.x.f_1\}, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P)$$
$$R = P [\![(\{\mathsf{if}\ g_2\ \mathsf{then}\ \mathsf{assign}.x.f_2\}, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P)$$

and $x \in \mathbf{w}P$. Let $\mathfrak{R}$ be either unless or $\mathsf{ensures}$. If J satisfies $[J \Rightarrow (\lambda s.\ f_1.s, g_1.s = f_2.s, g_2.s)]$ then we have:

$$\frac{{}_Q\vdash J \wedge p\ \mathfrak{R}\ q}{{}_R\vdash J \wedge p\ \mathfrak{R}\ q} \quad \text{and} \quad \frac{J\ {}_Q\vdash p \rightarrowtail q}{J\ {}_R\vdash p \rightarrowtail q}$$

◀

As an example consider two programs P and Q which communicate through the assignment (of P) $\mathsf{if}\ g\ \mathsf{then}\ y := x$ where x is intended to be a variable of P and y of Q. The assignment can be viewed as an action by P to send a new datum it keeps in

[9] A stronger result was given by Singh in [Sin93].

x to the variable y. As it is, P can send a new datum whenever g is true. Suppose that we want to implement this communication on a synchronous machine. That is, sending a new datum is only possible if not only P, but also Q is ready to receive the datum. Below we show an implementation of this. The read, write, and init sections are omitted for the sake of simplicity.

```
prog    $P'$
assign
        ...
[]      if Prdy $\wedge$ Qrdy then if $g$ then $y$, Prdy := $x$, false
[]      if $\neg$Prdy $\wedge$ $\neg$Qrdy then Prdy := true

prog    $Q'$
assign
        ...
[]      if Prdy $\wedge$ $\neg$Qrdy then Qrdy := true
[]      if $\neg$Prdy $\wedge$ Qrdy then Qrdy := false
```

The variables Prdy and Qrdy are assumed fresh with respect to P and Q. P' is ready to send a new datum if Prdy is true and Q' is ready to receive one if Qrdy is true. Only when Prdy and Qrdy are both true then a communication can take place. Notice that the protocol is in fact the 4-phase hand-shake protocol as in Figure 4.14.

We feel that $P' [] Q'$ somehow 'implements' $P [] Q$. But how can one justify this? We observe that we can obtain $P' [] Q'$ from P and Q through a series of previously described transformations, and recall that the transformations preserve —under some conditions— unless and $\rightarrowtail$ properties.

First, we can transform P by adding an assignment Prdy := true and extending the action if g then $y := x$ with if g then Prdy := false. Notice that these are assignments to Prdy, which is fresh. We obtain the following program:

```
prog    $P_1$
assign
        ...
[]      (if $g$ then Prdy := false) ; (if $g$ then $y := x$)
[]      Prdy := true
```

By adding assignments to the fresh variable Qrdy we can also transform Q to the following:

```
prog    $Q_1$
assign
        ...
[]      Qrdy := true
[]      Qrdy := false
```

By adding guards we can obtain P' from P_1 and Q' from Q_1.

By the transformation laws given earlier, it can be concluded that ${}_{P[]Q}\vdash p$ **unless** q implies ${}_{P'[]Q'}\vdash p$ **unless** q and $J\ {}_{P[]Q}\vdash p \rightarrowtail q$ implies $J\ {}_{P'[]Q'}\vdash p \rightarrowtail q$, if J, p, and q are all confined by Pred.($\mathbf{r}P$) and if each new guard eventually holds and can only be falsified by the action it stands guard for. If the reader observes the code of P' and Q' and considers the fact that Pry and Qrdy are fresh variables, he should be able to conclude that the latter condition is met.

One can continue —or take different transformations— by, for example, sharpening the condition that determines the readiness of Q to receive and hence giving Q more control in synchronizing with P.

4.10 The Semantics of UNITY

Eventually, one may want to relate the logic defined by UNITY and some operational semantics. In doing so, one usually hopes to investigate how far the logic reflects the 'real' world. This raises the question of soundness and completeness of the logic with respect to the given operational semantics. Another reason is that in some cases, reasoning may be easier if conducted at the operational level. So, some ability to go back and forth between the logical and operational level will be appreciated.

In this section, an operational semantics for UNITY will be given. The semantic domains are quite straightforwardly chosen, namely all possible sequences of states which can be generated by a UNITY program (this is a standard model). An operational notion of unless and $\mapsto$ (leads-to) will be defined based on these semantics. It has been proven that these two operators are *sound* with respect to their operational counterparts, but not complete. It has been shown that Sanders' version of unless and $\mapsto$ are *complete* [San91, Pac92]. There is no reason for us to repeat these results. However, we wish to mention here that we have *mechanically verified* the soundness results. Quite unfortunately, due to time constraints, we did not succeed in verifying the completeness results.

Recall that an execution of a UNITY program is an infinite sequence of actions such that each action occurs infinitely often (fairness). Let exec.P denote the set of all possible UNITY execution of P. We will represent an infinite sequence over A with a function from $\mathbb{N}$ to A. Let tr be the set of all possible sequence of states which can be generated by the executions in exec. A member of tr.P is called a *trace* of P.

Definition 4.10.1 Execution Set *EXEC_DEF*

$$\sigma \in \mathsf{exec}.P \;=\; (\forall i :: \sigma.i \in \mathbf{a}P) \;\wedge\; (\forall i, a : a \in \mathbf{a}P : (\exists j : i \leq j : \sigma.j = a))$$

Definition 4.10.2 Trace Set *TRACE_DEF*

$$\tau \in \mathsf{tr}.P \;=\; \mathbf{ini}P.(\tau.0) \;\wedge\; (\exists \sigma : \sigma \in \mathsf{exec}.P : (\forall i :: (\sigma.i).(\tau.i).(\tau.(i+1))))$$

◀

Let ${}_P\!\vdash p \,\mathfrak{U}\, q$ mean that for any trace τ of P, if $\tau.i$ satisfies $p \wedge \neg q$, then $\tau.(i+1)$ satisfies $p \vee q$. Let ${}_P\!\vdash p \,\mathfrak{L}\, q$ mean that for any trace τ of P, if $\tau.i$ satisfies p, then there exists a j, $i \leq j$, such that $\tau.j$ satisfies q. Indeed, $\mathfrak{U}$ and $\mathfrak{L}$ are intended to be the operational interpretation of unless and $\mapsto$.

Definition 4.10.3 Operational Unless *tUNLESS_ADEF1*

$$_P\vdash p\,\mathfrak{U}\,q \;=\; (\forall i,\tau : \tau \in \mathsf{tr}.P : (p \wedge \neg q).(\tau.i) \;\Rightarrow\; (p \vee q).(\tau.(i+1)))$$

Definition 4.10.4 Operational Leads-to *tLEADSTO_ADEF1*

$$_P\vdash p\,\mathfrak{L}\,q \;=\; (\forall i,\tau : \tau \in \mathsf{tr}.P : p.(\tau.i) \;\Rightarrow\; (\exists j : i \leq j : q.(\tau.j)))$$

---

unless and $\mapsto$ are sound with respect to the above operational interpretation. They are however not *complete*. The Sanders' definition of unless and $\mapsto$ (given in Section 4.6) are complete with respect to the above interpretation. The soundness of Sanders' definition follows from the soundness of the standard unless and $\mapsto$. We did not verify any completeness result. If the reader is interested, an elegant completeness proof can be found in [Pac92].

Theorem 4.10.5 Soundness of unless *UNLESS_IMP_tUNLESS*

$$(\,_P\vdash p \text{ unless } q) \;\Rightarrow\; (\,_P\vdash p\,\mathfrak{U}\,q)$$

Theorem 4.10.6 Soundness of $\mapsto$ *LEADSTO_IMP_tLEADSTO*

$$(\,_P\vdash p \mapsto q) \;\Rightarrow\; (\,_P\vdash p\,\mathfrak{L}\,q)$$

Now, how about the operational meaning of $\rightarrowtail$? We could not come with any satisfactory answer. The only thing that we know is, as given equation $(4.5.3)_{54}$ in Section 4.5, that $\rightarrowtail$ includes $\mapsto$:

$$(\mathsf{true}\;_P\vdash p \rightarrowtail q) \;\Rightarrow\; (\,_P\vdash p \mapsto q)$$

Or, slightly more general, we can prove:

$$(J\;_P\vdash p \rightarrowtail q) \;\Rightarrow\; (\,_P\vdash J \wedge p \mapsto q) \qquad (4.10.1)$$

Since $J\;_P\vdash p \rightarrowtail q$ also implies $_P\vdash J$ unless false, It follows then, that $J\;_P\vdash p \rightarrowtail q$ implies $_P\vdash J\,\mathfrak{U}\,\mathsf{false}$ and $_P\vdash J \wedge p\,\mathfrak{L}\,q$. We do not expect equivalence though, because $\rightarrowtail$ is quite different from $\mapsto$. This has been suggested early in Section 4.5, but let us go over an example presented there again. Consider the following program:

```
prog    $P$
read    $\{a, x\}$
write   $\{x\}$
init    true
assign  if $a = 0$ then $x := 1$
[]      if $a = 1$ then $x := 1$
[]      if $a = 2$ then $x := x + 1$
```

In the above program we have $(b = 0) \wedge a < 2 \mapsto (x = 1)$. We expect that the $\rightarrowtail$ version of this property, namely $(b = 0) \wedge a < 2 \vdash \textsf{true} \rightarrowtail (x = 1)$, also holds. But this is not true. The property $(b = 0) \wedge a < 2 \mapsto (x = 1)$ can be concluded because we have $(b = 0) \wedge (a = 0)$ ensures $(x = 1)$ and $(b = 0) \wedge (a = 1)$ ensures $(x = 1)$ and we can join them using the disjunctivity of $\mapsto$. Unfortunately we cannot do the same with $\rightarrowtail$. If we do that, we will get an unsound logic. Consider the program TikToe in Figure 4.9. It is redisplayed below:

```
prog    TikToe
read    {a, b}
write   {a}
init    true
assign  if a = 0 then a := 1
[]      if a = 1 then a := 0
[]      if b ≠ 0 then a := a + 1
```

The programs P and TikToe are write-disjoint. Suppose $(b = 0) \wedge a < 2 \; {}_P\!\vdash \textsf{true} \rightarrowtail (x = 1)$ holds. The predicate $(b = 0) \wedge a < 2$ is also stable in Tiktoe. By the TRANSPARENCY$_{72}$ principle we conclude that $(b = 0) \wedge a < 2 \; {}_{P[]\mathsf{TikToe}}\!\vdash \textsf{true} \rightarrowtail (x = 1)$ also holds. But this simply cannot be true. Consider the execution:

```
[ if a = 0 then a := 1 ; if a = 0 then x := 1 ;
  if a = 1 then a := 0 ; if a = 1 then x := 1 ;
  if a = 2 then x := x + 1 ; if b ≠ 0 then a := a + 1 ]*
```

which is a fair execution of $P[]$TikToe, but with this execution x will never be equal to 1 if initially $x \neq 1 \wedge a < 2 \wedge (b = 0)$.

So, $\rightarrowtail$ is not as disjunctive as $\mapsto$ can be: it has to be less disjunctive because, as we have seen, this is crucial for the TRANSPARENCY law. As said, we could not come up with a satisfactory operational semantics for $(\lambda p, q.\; J \; {}_P\!\vdash p \rightarrowtail q)$. We suspect however, that this is the largest subrelation of $(\lambda p, q.\; {}_P\!\vdash J \wedge p \mathfrak{L} q)$ which satisfies the TRANSPARENCY law.

4.11 Related Work

In [UHK94], Udink, Herman, and Kok defined a new progress operator. The new operator is somewhere between ensures and $\mapsto$. It has a very nice compositionality property but it is a rather complicated operator. However, the authors also provided a class of program transformations which preserve safety and progress under the new operator. Reasoning can be carried out in terms of transformations. In fact, the transformations discussed in Section 4.9 were much inspired by the work in [UHK94].

The issue of compositionality in distributed programming has received quite a lot of attention. In [Zwi88] Zwiers proposed a compositional logic for synchronously communicating processes. In [dBvH94] de Boer and van Hulst proposed a compositional logic for asynchronous systems, and in [PJ91] Pandya and Joseph proposed yet another compositional logic for both synchronous and asynchronous systems. The focus of these papers are focussed around the assumed means of communication, namely channels. Partial correctness is considered. In UNITY however, total correctness is very important, since otherwise no progress can be concluded. In this thesis, attention is focused on the compositionality of progress in general, but indeed further research is required to develop some basic theory for UNITY regarding channel-based communication. Such an investigation will surely benefit the results of the above mentioned papers.

Closely related work was done by de Boer and his colleagues in [dBKPJ93] where they gave a compositional semantics of local blocks. A local block is a part of a program in which it does some internal computation. Such a computation is not visible from outside, and therefore cannot be directly influenced either. Recall that in Section 4.8 we discussed write-disjoint programs. If two programs P and Q are write-disjoint, then the write variables of P are in a sense local, because they cannot be written by Q — although Q may still be able to observe them. In [UK93a] Udink and Kok investigated the relation between various operational semantics for UNITY and the preservation of UNITY properties under program refinement. In their subsequent paper [UK93b], semantics that preserves program refinement within a context were proposed.

4.12 Postponed Proofs

Theorem 4.9.1
Let P and Q be UNITY programs and $f \in \mathsf{Action}{\to}\mathsf{Action}$ such that: $\mathbf{a}Q = \{f.a; a \mid a \in \mathbf{a}P\} \cup \{f.a \mid a \in A\}$ for some $A \subseteq \mathsf{Action}$. Let $\mathfrak{R}$ be either unless or $\mathsf{ensures}$. We have:

$$\frac{(\forall a : a \in \mathbf{a}P : \mathbf{r}P \not\leftarrow f.a) \;\wedge\; ({}_P\!\vdash p \;\mathfrak{R}\; q)}{{}_Q\!\vdash (p \restriction \mathbf{r}P) \;\mathfrak{R}\; (q \restriction \mathbf{r}P)}$$

Proof:
We will only show the case $\mathfrak{R} = \mathsf{unless}$. The case of $\mathsf{ensures}$ can be proven in a much similar way. We have to show:

$${}_Q\!\vdash (p \restriction \mathbf{r}P) \;\mathsf{unless}\; (q \restriction \mathbf{r}P)$$

By the definition$_{42}$ of unless it suffices to show that for all $b \in \mathbf{a}Q$ we have:

$$\{(p \restriction \mathbf{r}P) \wedge \neg(q \restriction \mathbf{r}P)\}\; b\; \{(p \restriction \mathbf{r}P) \vee (q \restriction \mathbf{r}P)\}$$

If $b = f.a; a$, for some $a \in \mathbf{a}P$, we derive:

$$\{(p \restriction \mathbf{r}P) \wedge \neg(q \restriction \mathbf{r}P)\}\; f.a; a\; \{(p \restriction \mathbf{r}P) \vee (q \restriction \mathbf{r}P)\}$$

$=$ { projection distributes over predicate operators }

$$\{(p \wedge \neg q) \restriction \mathbf{r}P\}\; f.a; a\; \{(p \vee q) \restriction \mathbf{r}P\}$$

$\Leftarrow$ { Corollary 3.4.4$_{32}$ }

$$
\begin{array}{ll}
& \mathbf{r}P \not\leftarrow f.a \;\wedge\; (\mathbf{r}P)^{\mathsf{c}} \not\rightarrow a \;\wedge\; \{p \wedge \neg q\}\; a\; \{p \vee q\} \\
= & \{\ P \text{ is a UNITY program }\} \\
& \mathbf{r}P \not\leftarrow f.a \;\wedge\; \{p \wedge \neg q\}\; a\; \{p \vee q\} \\
= & \{\ \text{assumption }\} \\
& \{p \wedge \neg q\}\; a\; \{p \vee q\}
\end{array}
$$

The last follows from ${}_P\!\vdash p\ \mathsf{unless}\ q$.

If $b = f.a$, for some $a \in A$, then it can also be written as $f.a;\mathsf{skip}$ and an argument quite similar to the one applies.

▲

Theorem 4.9.4
Let P and Q be UNITY programs, $g \in \mathsf{Action}{\rightarrow}\mathsf{Pred}$, and $A \subseteq \mathbf{a}P$ such that $\mathbf{w}P \subseteq \mathbf{w}Q$ and $\mathbf{a}Q = (\mathbf{a}P \backslash A) \ \cup\ \{\mathsf{if}\ g.a\ \mathsf{then}\ a \mid a \in A\}$. Let Q_{-a} be the same program as Q, except that the action a is deleted. We have:

$$
\frac{(\forall a : a \in A : ({}_{Q_{-a}}\!\vdash \circlearrowleft J \wedge g.a) \;\wedge\; (J\ {}_Q\!\vdash \mathsf{true} \rightarrowtail g.a)) \;\wedge\; (J\ {}_P\!\vdash p \rightarrowtail q)}{J\ {}_Q\!\vdash p \rightarrowtail q}
$$

Proof:
Using $\rightarrowtail$ INDUCTION$_{56}$ it suffices to show that $\mathfrak{R} = (\lambda p, q.\ J\ {}_Q\!\vdash p \rightarrowtail q)$ is transitive, left-disjunctive, and includes $\mathfrak{E} = (\lambda p, q.\ J\ {}_P\!\vdash p\ \mathfrak{ensures}\ q)$, assuming that ${}_{Q_{-a}}\!\vdash \circlearrowleft J \wedge g.a$ and $J\ {}_Q\!\vdash \mathsf{true} \rightarrowtail g.a$ hold for all $a \in A$.

The transitivity and left-disjunctivity of $\mathfrak{R}$ follow from $\rightarrowtail$ TRANSITIVITY$_{56}$ and DISJUNCTION$_{56}$. As for the inclusion of $\mathfrak{E}$, assume $\mathfrak{E}.p.q$. Hence, by the definitions of $\mathfrak{E}$, $\mathfrak{ensures}$, and $\mathsf{ensures}$ we have:

$$p, q \in \mathsf{Pred}.(\mathbf{w}P) \tag{4.12.1}$$

$${}_P\!\vdash \circlearrowleft J \tag{4.12.2}$$

$${}_P\!\vdash J \wedge p\ \mathsf{unless}\ q \tag{4.12.3}$$

$$\{J \wedge p \wedge \neg q\}\; a\; \{q\} \tag{4.12.4}$$

for some $a \in \mathbf{a}P$. If $a \notin A$ (and hence $a \in \mathbf{a}Q$) we derive:

$$
\begin{array}{ll}
& J\ {}_Q\!\vdash p \rightarrowtail q \\
\Leftarrow & \{\ \rightarrowtail \text{INTRODUCTION}_{56}\ \} \\
& p, q \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; ({}_Q\!\vdash \circlearrowleft J) \;\wedge\; ({}_Q\!\vdash J \wedge p\ \mathsf{ensures}\ q) \\
= & \{\ \mathbf{w}P \subseteq \mathbf{w}Q,\ \text{CONFINEMENT MONOTONICITY}_{30},\ (4.12.1)\ \} \\
& ({}_Q\!\vdash \circlearrowleft J) \;\wedge\; ({}_Q\!\vdash J \wedge p\ \mathsf{ensures}\ q) \\
\Leftarrow & \{\ \text{definition of } \mathsf{ensures},\ a \in \mathbf{a}Q\ \} \\
& ({}_Q\!\vdash \circlearrowleft J) \;\wedge\; ({}_Q\!\vdash J \wedge p\ \mathsf{unless}\ q) \;\wedge\; \{J \wedge p \wedge \neg q\}\; a\; \{q\} \\
= & \{\ (4.12.4)\ \} \\
& ({}_Q\!\vdash \circlearrowleft J) \;\wedge\; ({}_Q\!\vdash J \wedge p\ \mathsf{unless}\ q)
\end{array}
$$

$$\begin{array}{ll}
\Leftarrow & \{ \text{ definition}_{44} \text{ of } \circlearrowleft, \text{ Theorem } 4.9.3_{76} \} \\
& ({}_P\vdash \circlearrowleft J) \;\wedge\; ({}_P\vdash J \wedge p \;\mathsf{unless}\; q)
\end{array}$$

The first is (4.12.2) and the second is (4.12.3). If $a \in A$ we derive first:

$$\begin{array}{ll}
& J \;{}_Q\vdash p \rightarrowtail q \\
\Leftarrow & \{ \rightarrowtail \textsc{Cancellation}_{56} \} \\
& q \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; (J \;{}_Q\vdash p \rightarrowtail (p \wedge g.a) \vee q) \;\wedge\; (J \;{}_Q\vdash p \wedge g.a \rightarrowtail q) \\
\Leftarrow & \{ \rightarrowtail \mathrm{PSP}_{56} \} \\
& p, q \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; ({}_Q\vdash J \wedge p \;\mathsf{unless}\; q) \;\wedge\; (J \;{}_Q\vdash \mathsf{true} \rightarrowtail g.a) \;\wedge\; (J \;{}_Q\vdash p \wedge g.a \rightarrowtail q) \\
= & \{ \mathbf{w}P \subseteq \mathbf{w}Q, \textsc{Confinement Monotonicity}_{30}, (4.12.1) \} \\
& ({}_Q\vdash J \wedge p \;\mathsf{unless}\; q) \;\wedge\; (J \;{}_Q\vdash \mathsf{true} \rightarrowtail g.a) \;\wedge\; (J \;{}_Q\vdash p \wedge g.a \rightarrowtail q) \\
\Leftarrow & \{ \text{ Theorem } 4.9.3_{76} \} \\
& ({}_P\vdash J \wedge p \;\mathsf{unless}\; q) \;\wedge\; (J \;{}_Q\vdash \mathsf{true} \rightarrowtail g.a) \;\wedge\; (J \;{}_Q\vdash p \wedge g.a \rightarrowtail q)
\end{array}$$

The first conjunct is (4.12.3) and the second is an assumption. Let Q_a be defined as:

$$Q_a = (\{\mathsf{if}\; g.a \;\mathsf{then}\; a\}, \mathbf{ini}Q, \mathbf{r}Q, \mathbf{w}Q)$$

Note that $Q = Q_{-a} [\!] Q_a$. For the third conjunct we derive:

$$\begin{array}{ll}
& J \;{}_Q\vdash p \wedge g.a \rightarrowtail q \\
\Leftarrow & \{ \rightarrowtail \textsc{Introduction}_{56}, \text{ confinement is preserved by } \wedge \} \\
& p, g.a, q \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; ({}_Q\vdash \circlearrowleft J) \;\wedge\; ({}_Q\vdash J \wedge p \wedge g.a \;\mathsf{ensures}\; q) \\
= & \{ \mathbf{w}P \subseteq \mathbf{w}Q, \textsc{Confinement Monotonicity}_{30}, (4.12.1) \} \\
& g.a \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; ({}_Q\vdash \circlearrowleft J) \;\wedge\; ({}_Q\vdash J \wedge p \wedge g.a \;\mathsf{ensures}\; q) \\
\Leftarrow & \{ \text{ Theorem } 4.9.3_{76}, \text{ definition}_{44} \text{ of } \circlearrowleft, (4.12.2) \} \\
& g.a \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; ({}_Q\vdash J \wedge p \wedge g.a \;\mathsf{ensures}\; q) \\
\Leftarrow & \{ Q = Q_{-a} [\!] Q_a, \;\mathsf{ensures}\; \textsc{Composition}_{60} \} \\
& g.a \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; ({}_{Q_{-a}}\vdash J \wedge p \wedge g.a \;\mathsf{unless}\; q) \;\wedge\; ({}_{Q_a}\vdash J \wedge p \wedge g.a \;\mathsf{ensures}\; q) \\
\Leftarrow & \{ \mathsf{unless}\; \textsc{Simple Conjunction}_{46}, \text{ definition}_{44} \text{ of } \circlearrowleft \} \\
& g.a \in \mathsf{Pred}.(\mathbf{w}Q) \wedge ({}_{Q_{-a}}\vdash J \wedge p \;\mathsf{unless}\; q) \wedge ({}_{Q_{-a}}\vdash \circlearrowleft J \wedge g.a) \wedge ({}_{Q_a}\vdash J \wedge p \wedge g.a \;\mathsf{ensures}\; q) \\
= & \{ \mathbf{a}Q_{-a} \subseteq \mathbf{a}Q, \text{ assumption } \} \\
& g.a \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; ({}_Q\vdash J \wedge p \;\mathsf{unless}\; q) \;\wedge\; ({}_{Q_a}\vdash J \wedge p \wedge g.a \;\mathsf{ensures}\; q) \\
= & \{ \text{ Theorem } 4.9.3_{76}, (4.12.3) \} \\
& g.a \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; ({}_{Q_a}\vdash J \wedge p \wedge g.a \;\mathsf{ensures}\; q) \\
= & \{ \text{ definition of } \mathsf{ensures} \text{ and } Q_a \} \\
& g.a \in \mathsf{Pred}.(\mathbf{w}Q) \;\wedge\; \{J \wedge p \wedge g.a \wedge \neg q\} \;\mathsf{if}\; g.a \;\mathsf{then}\; a \;\{q\}
\end{array}$$

The first conjunct follows from $J \;{}_Q\vdash \mathsf{true} \rightarrowtail g.a$ and $\rightarrowtail$ $\textsc{Confinement}_{57}$. For the second, we derive:

$$\begin{array}{ll}
& \{J \wedge p \wedge g.a \wedge \neg q\} \;\mathsf{if}\; g.a \;\mathsf{then}\; a \;\{q\} \\
= & \{ \text{ definition Hoare triple and } \mathsf{if}\; g.a \;\mathsf{then}\; a \;\}
\end{array}$$

$$
\begin{array}{ll}
& (\forall s,t :: (J \land p \land g.a \land \neg q).s \land (g.a.s \Rightarrow a.s.t) \land (\neg g.a.s \Rightarrow s = t) \;\Rightarrow\; q.t) \\
= & \quad \{ \text{ definition of predicate operators } \} \\
& (\forall s,t :: (J \land \land p \land \neg q).s \land g.a.s \land (g.a.s \Rightarrow a.s.t) \land (\neg g.a.s \Rightarrow s = t) \;\Rightarrow\; q.t) \\
\Leftarrow & \quad \{ \text{ predicate calculus } \} \\
& (\forall s,t :: (J \land p \land \neg q).s \land a.s.t \;\Rightarrow\; q.t) \\
= & \quad \{ \text{ definition of Hoare triple } \} \\
& (4.12.4)
\end{array}
$$

▲

Theorem 4.9.5
Let P, Q and R be UNITY programs such that

$$
\begin{array}{l}
Q = P[\!](\{ \mathsf{if}\ g_1\ \mathsf{then\ assign}.x.f_1 \}, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P) \\
R = P[\!](\{ \mathsf{if}\ g_2\ \mathsf{then\ assign}.x.f_2 \}, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P)
\end{array}
$$

and $x \in \mathbf{w}P$. Let $\mathfrak{R}$ be either unless or $\mathsf{ensures}$. If J satisfies $[J \Rightarrow (\lambda s.\ f_1.s, g_1.s = f_2.s, g_2.s)]$ then we have:

$$
\frac{{}_{Q}\!\vdash J \land p \,\mathfrak{R}\, q}{{}_{R}\!\vdash J \land p \,\mathfrak{R}\, q} \quad \text{and} \quad \frac{J \;{}_{Q}\!\vdash p \rightarrowtail q}{J \;{}_{R}\!\vdash p \rightarrowtail q}
$$

▶

Proof:
We will only proof the unless case. The $\mathsf{ensures}$ case can be proven in much the same way and the $\rightarrowtail$ case can subsequently be proven easily using $\rightarrowtail$ INDUCTION$_{56}$. It suffices to show that:

$$\{J \land p \land \neg q\}\ b\ \{(J \land p) \lor q\}$$

holds for all $b \in \mathbf{a}R$. If $b \in \mathbf{a}P$ then it is trivially implied by ${}_{Q}\!\vdash J \land p$ unless q. If $b = \mathsf{if}\ g_2\ \mathsf{then\ assign}.x.f_2$ we derive:

$$
\begin{array}{ll}
& \{J \land p \land \neg q\}\ \mathsf{if}\ g_2\ \mathsf{then\ assign}.x.f_2\ \{(J \land p) \lor q\} \\
= & \quad \{ \text{ definition of Hoare triple, } \mathsf{if\text{-}then} \text{ construct, and assignment } \} \\
& (\forall s,t ::\ (J \land p \land \neg q).s \;\land\; (g_2.s \Rightarrow (t.x = f_2.s) \land (t \restriction \{x\}^{\mathsf{c}} = s \restriction \{x\}^{\mathsf{c}})) \;\land \\
& \qquad\qquad (\neg g_2.s \Rightarrow (s = t)) \;\Rightarrow\; ((J \land p) \lor q).t) \\
= & \quad \{ \text{ definition of predicate operators } \} \\
& (\forall s,t ::\ J.s \;\land\; (p \land \neg q).s \;\land\; (g_2.s \Rightarrow (t.x = f_2.s) \land (t \restriction \{x\}^{\mathsf{c}} = s \restriction \{x\}^{\mathsf{c}})) \;\land \\
& \qquad\qquad (\neg g_2.s \Rightarrow (s = t)) \;\Rightarrow\; ((J \land p) \lor q).t) \\
= & \quad \{ [J \Rightarrow (\lambda s.\ f_1.s, g_1.s = f_2.s, g_2.s)] \} \\
& (\forall s,t ::\ J.s \;\land\; (p \land \neg q).s \;\land\; (g_1.s \Rightarrow (t.x = f_1.s) \land (t \restriction \{x\}^{\mathsf{c}} = s \restriction \{x\}^{\mathsf{c}})) \;\land \\
& \qquad\qquad (\neg g_2.s \Rightarrow (s = t)) \;\Rightarrow\; ((J \land p) \lor q).t) \\
= & \quad \{ \text{ definition of Hoare triple and } \mathsf{if\text{-}then} \text{ construct, assignment } \} \\
& \{J \land p \land \neg q\}\ \mathsf{if}\ g_1\ \mathsf{then\ assign}.x.f_1\ \{(J \land p) \lor q\}
\end{array}
$$

▲

Nothing is ever ordinary!
You may think that the space is empty
—that there is nothing in it
But your eyes are not perfect.
—*Srila Prabhupada in Life Comes from Life*

Chapter 5 Stabilization

A self-stabilizing system is, roughly speaking, a system which is capable to recover from arbitrary transient failures. Such a property is obviously very useful, although one may find the requirement to consider arbitrary failures to be too strong. A more general notion called convergence is useful to express a more restricted form of stabilizing systems. In this chapter a UNITY definition of convergence will be presented, together with a set of calculational laws. The laws are used to derive Lentfert's Fair and Successive Approximation (FSA) algorithm.

5.1 Introduction

THE notion of self-stabilization was first introduced by Dijkstra in [Dij74]. In his paper Dijkstra considers a network of processes. A *central daemon* is introduced to schedule the execution of the processes and there is a notion of *privilege*. Only privileged processes may be executed, but during the execution, the privileges may indeed change. There is also a set of pre-defined *legitimate* states describing the objective of the system. Dijkstra defines self-stabilization as:

> *Regardless of the initial state and regardless of the privilege (*privileged process*) selected each time for the next move, at least one privilege (*privileged process*) will always be present and the system is guaranteed to find itself in a legitimate state after a finite number of moves.*

and in addition:

> *In each legitimate state each possible move will bring the system again in a legitimate state.*

Since Dijkstra's paper, there have been many advances in computer science research which made it possible to reason about distributed systems at a more abstract level. People re-discovered temporal logics and with such a logic at hand it is no longer necessary to reason operationally in terms of central daemon and privileges (which basically define the enabledness of an action). Most importantly though, reasoning about self-stabilization can now be carried out completely within such a logic. The idea of self-stabilization itself is first formalized by Arora and Gouda in [AG90], but their reasoning is still done informally. A step forward is made by Herman in [Her91] by proposing a number of compositional laws of stabilization. A truly formal treatment of stabilization is later given by Lenfert and Swierstra in [LS93] in which the concept of stabilization in UNITY is formalized and various calculational properties of stabilization are proved.

To re-phrase Dijkstra's definition, self-stabilization requires two things. First, the system in question is required to reach a legitimate state in a finite time, regardless of the initial state of the system. Second, the system will subsequently remain within the space of legitimate states. The first requirement means progress, and the second stability. If the set of legitimate states is described by a state-predicate q, and P is a program which self-stabilizes to q, then in UNITY we can express this as follows:

$$({}_P\vdash \mathsf{true} \mapsto q) \;\wedge\; ({}_P\vdash \circlearrowleft q) \tag{5.1.1}$$

Notice the pre-condition **true** in the progress part of the above specification. It means that P can progress to q regardless of its initial state.

What about the failure-recovery property of a self-stabilizing system we mentioned earlier? As a self-stabilizing system P tries to reach its objective, its environment may be unstable: it may produce some transient errors, or undergo a spontaneous reconfiguration, which affect the consistency of the variables upon which P depends. Because of this 'malicious' behavior, such an environment is sometimes called an *adversary*. For example, Dijkstra's central daemon can be considered as an adversary due to its non-deterministic behavior. If an error occurs, one can consider the state after the error as a new initial state. Since the system can reach its objective regardless of its initial state, it will do so too now from the new state. In fact, the fact that a program can tolerate arbitrary transient errors and the fact that it need not to be initialized are equivalent.

Self-stabilization is a strong design goal. Perhaps too strong as it may be either too difficult to achieve or only be achievable at the expense of other goals. As failures are not always arbitrary it is useful to consider recovery from a restricted set of failures. The notion of self-stabilization can be generalized to express this kind of weaker recovery. There is another reason to make a generalization: it may yield more attractive and useful calculational laws. In [Len93] Lenfert used the notion of *eventually-implies*. In a program P, p is said to eventually imply q if:

$$({}_P\vdash p \mapsto q \vee \neg p) \;\wedge\; ({}_P\vdash p \wedge q \;\mathsf{unless}\; \neg p) \tag{5.1.2}$$

In particular, if p holds and remains stable, then the above implies that P will progress to q, which then will remain to hold. It is more general because P does not have to stabilize to q from *all* possible initial states, but only those states satisfying p.

We will however take a less radical approach. Basically what we will do is simply to replace the **true** in (5.1.1) with an some predicate p. So, given p, a program P is said to stabilize to q if:

$$({}_P\vdash p \mapsto q) \;\wedge\; ({}_P\vdash \circlearrowleft q) \tag{5.1.3}$$

This corresponds with the notion of *convergence* introduced by Arora and Gouda in [AG92]. Another closely related concept is *adaptiveness* introduced by Gouda and

Herman in [GH91]. Recall that our goal is to mechanically verify the work of Lentfert on self-stabilizing minimal distance algorithms [Len93]. Convergence turns out to satisfy all calculation laws used by Lentfert, and some more. In addition, it is also a very intuitive concept. These are the reasons that we choose convergence above eventually-implies.

5.2 Convergence

The definition of stabilization or convergence as in (5.1.3) is what by Lentfert in [Len93] is called *leads-to-a-stable*. There is still a small problem with the definition in (5.1.3). Suppose that a program P can progress from p to q. However, P may not remain in q immediately after the first time q holds. Instead, P may need several iterations before it finally remains within q. This can be encoded by requiring P to converge to a stronger (than q) predicate. This predicate does not need to be fully described. It suffices to know that it implies q[1]. So, here is the definition of convergence (in terms of $\rightarrowtail$ instead of $\mapsto$):

Definition 5.2.1 Convergence *CON*

$$J \;{}_P\!\vdash p \rightsquigarrow q$$
$$=$$
$$q \in \mathsf{Pred}.(\mathbf{w}P) \;\wedge\; (\exists q' :: (J \;{}_P\!\vdash p \rightarrowtail q' \wedge q) \;\wedge\; ({}_P\!\vdash \circlearrowleft(J \wedge q' \wedge q)))$$

◀

So, $J \;{}_P\!\vdash p \rightsquigarrow q$ implies that under the stability of J, from p the program P will eventually find itself in a situation where q holds and will remain to hold. If it is obvious from the context which P and which J are referred, they will be often dropped from the formula. $J \vdash p \rightsquigarrow q$ is pronounced "given the stability of J, from p, P converges to q".

5.2.1 An Example: Leader Election

As an example, let us consider a derivation of a self-stabilizing program for choosing a 'leader' in a network of processes. The problem was first posed in [lL77]. Leader election has a lot of applications in distributed computing. For example, to appoint a central server when several candidates are available. The selection is required to be non-deterministic[2].

[1] This notion of convergence is what by Burns, Gouda, and Miller called *pseudo-stabilization* [BGM90].

[2] The reader may remark that as in [Tel94] we should also require that each process must run the same program. Note however that this kind of requirement is intended for non-self-stabilizing distributed systems, for it is known that no distributed, self-stabilizing system exists if the system is fully symmetric [Gou87, Sch93]

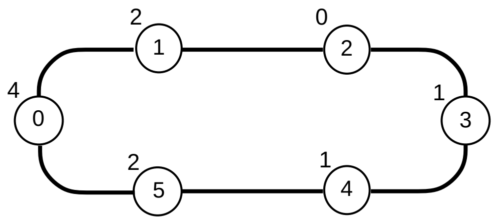


Figure 5.1: *A ring network.*

We have N processes numbered from 0 to $N-1$ connected in a ring. Process i is connected to process i^+ where $^+$ is defined as:

$$i^+ \;=\; (i+1) \;\mathsf{mod}\; N$$

Figure 5.1 shows such a ring of six processes.

Each process i has a local variable $x.i$ that contains a natural number less than N. For example, the numbers printed above the circles in Figure 5.1 show the values of the $x.i$'s of the corresponding processes. The problem is to make all processes agree on a common value of the $x.i$'s. The selected number is then the number of the 'leader' process, which is why the problem is called 'leader election'. The computation has to be self-stabilizing and non-deterministic. The latter means that if the program is re-run several times, it should not always select the same leader.

To solve this, first we extend the $x.i$'s to range over natural numbers and allow them to have arbitrary initial values. The problem is generalized to computing a common value of $x.i$'s. The identity of the leader can be obtained by applying $\mathsf{mod}\, N$ to the resulting common natural number.

Let us define a predicate Ok as follows.

$$\mathsf{Ok} \;=\; (\forall i : i < N : x.i = x.i^+)$$

The specification of the problem can be expressed as follows:

$$\mathsf{LS0}: \quad \mathsf{true} \;{}_{\mathsf{ring}}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{Ok}$$

Here is our strategy to solve the above. We let the value of $x.0$ decrease to a value which can no longer be *'affected'* by the value of other $x.i$'s —we choose to rule that only those $x.i$'s whose value is lower than $x.0$ *may* affect $x.0$. This value of $x.0$ is then propagated along the ring to be copied to each $x.i$ and hence we now have a common value of the $x.i$'s. Recall the BOUNDED PROGRESS$_{50}$ principle from Chapter 4. It states that any transitive and left-disjunctive relation $\rightarrow$ satisfies:

$$\frac{q \rightarrow q \;\wedge\; (\forall M :: p \wedge (m = M) \rightarrow (p \wedge (m \prec M)) \vee q)}{p \rightarrow q}$$

if $\prec$ is well-founded. We know that $\rightarrowtail$ is transitive and left-disjunctive, so it satisfies the principle above. The principle states that if a program will either establish q or decrease m, then eventually q will hold since $\prec$ is well-founded and hence m cannot be decreased forever. In fact, the above described strategy (either Ok is established or $x.0$ decreases) is an instance of this principle. Let us now apply the principle to LS0:

$$\begin{array}{ll}
 & \mathsf{true} \vdash \mathsf{true} \rightsquigarrow \mathsf{Ok} \\
\Leftarrow & \{ \text{ Definition of } \rightsquigarrow \} \\
 & (\mathsf{true} \vdash \mathsf{true} \rightarrowtail \mathsf{Ok}) \ \wedge \ (\vdash \circlearrowleft \mathsf{Ok}) \\
\Leftarrow & \{ \text{ the } \textsc{Bounded Progress} \text{ principle } \} \\
 & (\forall M :: \mathsf{true} \vdash (x.0 = M) \rightarrowtail (x.0 < M) \vee \mathsf{Ok}) \ \wedge \ (\vdash \circlearrowleft \mathsf{Ok}) \ \wedge \ (\mathsf{true} \vdash \mathsf{Ok} \rightarrowtail \mathsf{Ok}) \\
\Leftarrow & \{ \rightarrowtail \textsc{Introduction}_{56}, \ \mathsf{ensures} \text{ satisfies } p \ \mathsf{ensures} \ p \ \} \\
 & (\forall M :: \mathsf{true} \vdash (x.0 = M) \rightarrowtail (x.0 < M) \vee \mathsf{Ok}) \ \wedge \ (\vdash \circlearrowleft \mathsf{Ok}) \ \wedge \ \mathsf{Ok} \in \mathsf{Pred}.(\mathbf{w}(\mathsf{ring}))
\end{array}$$

Note that the requirement $\mathsf{Ok} \in \mathsf{Pred}.(\mathbf{w}(\mathsf{ring}))$ is met if $\mathbf{w}(\mathsf{ring})$ contains all $x.i$'s, $i < N$. The progress part of the last formula above states that the value of $x.0$ *must* decrease while Ok is not established. But if Ok is not yet established then there must be some i such that $x.i \neq x.0$. A naive solution is to send the minimum value of the initial $x.i$'s to $x.0$ but this results a deterministic program which always chooses the minimum value of the $x.i$'s as the common value. So, we will try something else. We let each process copy its $x.i$ to $x.i^+$. In this way the value of some $x.i$ which is smaller —not necessarily the smallest possible— than $x.0$, if one exists, will eventually reach process 0, or it will disappear. Of course it is possible that values larger than $x.0$ reach process 0 first, but in this case process 0 simply ignores these values.

Let now ts be defined as follows:

$$\mathsf{ts} \ = \ N \ - \ \mathsf{max}\{n \mid (n \leq N) \wedge (\forall i : i < n : x.i = x.0)\} \tag{5.2.1}$$

Roughly, ts is the length of the tail segment of the ring whose elements are still different from $x.0$. Note that according to the just described strategy the value of $x.0$ either remains the same or it decreases. If it does not decrease, it will be copied to $x.1$, then to $x.2$, and so on. In doing so ts will be decreased. Note that $\mathsf{ts} = 0$ implies Ok. This is, again, an instance of the Bounded Progress principle (either ring establishes Ok or ts decreases). Let us now see how the strategy described above is translated to the formal level (confinement conditions will be omitted —they are met if $\mathbf{w}(\mathsf{ring})$ contains all $x.i$'s):

$$\begin{array}{ll}
 & \mathsf{true} \vdash (x.0 = M) \rightarrowtail (x.0 < M) \vee \mathsf{Ok} \\
\Leftarrow & \{ \text{ the } \textsc{Bounded Progress} \text{ principle } \} \\
 & (\forall K : K < N : \mathsf{true} \vdash \ (x.0 = M) \wedge (\mathsf{ts} = K) \rightarrowtail \\
 & \qquad\qquad ((x.0 = M) \wedge (\mathsf{ts} < K)) \vee \mathsf{Ok} \vee (x.0 < M)) \\
\Leftarrow & \{ \rightarrowtail \textsc{Introduction}_{56} \ \} \\
 & (\forall K : K < N : \ \vdash \ (x.0 = M) \wedge (\mathsf{ts} = K) \ \mathsf{ensures} \\
 & \qquad\qquad ((x.0 = M) \wedge (\mathsf{ts} < K)) \vee \mathsf{Ok} \vee (x.0 < M))
\end{array}$$

So, to summarize, we come to the following refinement of LS0:

```
prog    ring
read    $\{x.i \mid i < N\}$
write   $\{x.i \mid i < N\}$
init    true
assign  if $x.(N-1) < x.0$ then $x.0 := x.(N-1)$
[]      $([]i : i < N-1 : x.(i+1) := x.i)$
```

Figure 5.2: *Leader election in a ring.*

For all $M \in \mathbb{N}$ and $K < N$:

LS1.a: $\circlearrowright$ Ok
LS1.b: $(x.0 = M) \wedge (\mathsf{ts} = K)$
ensures $((x.0 = M) \wedge (\mathsf{ts} < K)) \vee \mathsf{Ok} \vee (x.0 < M)$

In addition: $\{x.i \mid i < N\} \subseteq \mathbf{w}(\mathsf{ring})$.

LS1.a states that once the processes agree on a common value, they maintain this situation. LS1.b states that if a common value has not been found, then either the length of the tail segment should become smaller, which can be achieved by copying the value of $x.i$ to $x.i^+$, or $x.0$ should decrease.

Without further proof, a program that satisfies the above specifications is presented in Figure 5.2[3]

5.2.2 Properties of Convergence

Leader election is our first example of program derivation. During its derivation, its original convergence specification was broken into a progress part and a safety part. Reasoning was continued purely using the laws for $\rightarrowtail$ and $\circlearrowright$. Some part of the calculation can also be carried out in terms of convergence. Figure 5.3 lists properties of convergence. Notice that $\rightsquigarrow$ shares some important properties of $\rightarrowtail$. However, what distinguishes $\rightsquigarrow$ from $\rightarrowtail$ is that the first is conjunctive (Theorem 5.2.10) and the second is not. Figure 5.4 presents two properties of $\rightsquigarrow$ regarding write-disjoint composition. Notice that convergence also satisfies the TRANSPARENCY principle.

Compared with leads-to-a-stable [Len93], defined as in (5.1.3), $\rightsquigarrow$ has the SUBSTITUTION law. In particular, the post-condition of $\rightsquigarrow$ can be weakened as in Hoare triples. With leads-to-a-stable this is not possible. Compared with eventually-implies [Len93] —see (5.1.2)—, $\rightsquigarrow$ is transitive whereas eventually-implies is not.

[3] In the read and write sections of the program in Figure 5.2 "$\{x.i \mid i < N\}$" denotes a set of variables. Another notation which the reader is perhaps more familiar with is: x: array $[0 \ldots N)$ of Val

Theorem 5.2.2 CONVERGENCE IMPLIES PROGRESS *CON_IMP_REACH*

$$P, J : \frac{p \leadsto q}{p \rightarrowtail q}$$

Theorem 5.2.3 $\leadsto$ STABLE BACKGROUND *CON_IMP_STABLE*

$$(J \;{}_P\!\vdash p \leadsto q) \;\Rightarrow\; ({}_P\!\vdash \circlearrowleft J)$$

Theorem 5.2.4 $\leadsto$ CONFINEMENT *CON_CONF_LIFT*

$$P, J : \frac{p \leadsto q}{p, q \in \mathsf{Pred}.(\mathbf{w}P)}$$

Theorem 5.2.5 $\leadsto$ INTRODUCTION *CON_ENSURES_LIFT, CON_IMP_LIFT*

$$P, J : \frac{\begin{array}{c} p, q \in \mathsf{Pred}.(\mathbf{w}P) \;\wedge\; (\circlearrowleft J) \;\wedge\; (\circlearrowleft (J \wedge q)) \\ [p \wedge J \Rightarrow q] \;\vee\; (p \wedge J \;\mathsf{ensures}\; q) \end{array}}{p \leadsto q}$$

Theorem 5.2.6 $\leadsto$ SUBSTITUTION *CON_SUBST*

$$P, J : \frac{[J \wedge p \Rightarrow q] \;\wedge\; [J \wedge r \Rightarrow s] \;\wedge\; p, s \in \mathsf{Pred}.(\mathbf{w}P) \;\wedge\; (q \leadsto r)}{p \leadsto s}$$

Theorem 5.2.7 ACCUMULATION *CON_SPIRAL*

$$P, J : \frac{(p \leadsto q) \;\wedge\; (q \leadsto r)}{p \leadsto q \wedge r}$$

Theorem 5.2.8 $\leadsto$ TRANSITIVITY *CON_TRANS*

$$P, J : \frac{(p \leadsto q) \;\wedge\; (q \leadsto r)}{p \leadsto r}$$

Theorem 5.2.9 $\leadsto$ DISJUNCTION *CON_DISJ*

$$P, J : \frac{(\forall i : i \in W : p.i \leadsto q.i)}{(\exists i : i \in W : p.i) \leadsto (\exists i : i \in W : q.i)} \quad \text{if } W \neq \emptyset$$

Theorem 5.2.10 $\leadsto$ CONJUNCTION *CON_CONJ*
For all *non-empty* and *finite* sets W:

$$P, J : \frac{(\forall i : i \in W : p.i \leadsto q.i)}{(\forall i : i \in W : p.i) \leadsto (\forall i : i \in W : q.i)}$$

Theorem 5.2.11 $\leadsto$ STABLE SHIFT *CON_STABLE_SHIFT*

$$P : \frac{p' \in \mathsf{Pred}.\mathbf{w}P \wedge (\circlearrowleft J) \;\wedge\; (J \wedge p' \vdash p \leadsto q)}{J \vdash p' \wedge p \leadsto q}$$

◀

Figure 5.3: *Some basic properties of* $\leadsto$.

Theorem 5.2.12 $\leadsto$ TRANSPARENCY *CON_TRANSPARANT*

$$\frac{P \div Q \;\wedge\; ({}_{Q}\!\vdash \circlearrowleft J) \;\wedge\; (J \;{}_{P}\!\vdash p \leadsto q)}{J \;{}_{P[\!]Q}\!\vdash p \leadsto q}$$

Theorem 5.2.13 $\leadsto$ WRITE-DISJOINT CONJUNCTION *CON_CONJ_qWD*

For any *non-empty* and *finite* set W:

$$J : \frac{(\forall i,j : i,j \in W \wedge (i \neq j) : P.i \div P.j) \;\wedge\; (\forall i : i \in W : \;{}_{P.i}\!\vdash p.i \leadsto q.i)}{{}_{([\!]i:i\in Q:P.i)}\!\vdash (\forall i : i \in W.p.i) \leadsto (\forall i : i \in W : q.i)}$$

Figure 5.4: *Convergence under write-disjoint composition.*

5.3 Inductive Decomposition

In sequential programming, we have a law for decomposing a while-loop specification into the specifications of the loop's guard and body. In fact, the whole sequential programming relies basically on loops. In UNITY we do not have loops. At least not explicitly. Recall that any execution of a UNITY program is infinite and that each action must be executed infinitely often. In this sense a UNITY program is actually one large while-loop. A loop decomposition law is basically an induction theorem, used to break the progress specification of the whole loop into the specification of each iteration step. We have seen such a theorem before, namely the BOUNDED PROGRESS$_{50}$ law from Chapter 4. The law is a consequence of well-founded induction and is applicable to both $\rightarrowtail$ and $\leadsto$ —see Figure 5.5. In fact, well-founded induction is a standard technique to prove termination and the $\mapsto$ version of the BOUNDED PROGRESS law appears in [CM88] as a standard technique to prove progress.

A stronger induction principle exists for convergence. The principle exploits the conjunctivity of convergence —a property which is not enjoyed by $\rightarrowtail$ or $\mapsto$. Imagine a tree of processes. Each process collects the results of its sons, makes its own progress and then passes the result to its father. Consider a node n with sons l and m. Suppose l and m make progress to, respectively, q_l and q_m. The progress of l and m does not however combine to $q_l \wedge q_m$ because progress is not conjunctive. However, if processes l and m converge to q_l and q_m then we do have $q_l \wedge q_m$. The process n may therefore make a stronger assumption to establish its own progress. If each process n has the following convergence property:

$$(\forall m : \text{"}m \text{ is a proper descendant of } n\text{"} : q_m) \leadsto q_n$$

then by applying tree induction one may conclude that eventually the system will converge to $(\forall m : m \in A : q_m)$, where A is the set of all processes in the tree. The principle applies not only to trees, but to structures which can be ordered by well-founded relations.

Let in $\prec$ be a well-founded relation over A and $m \in \mathsf{State}{\to}A$.

Theorem 5.3.1 $\rightarrowtail$ BOUNDED PROGRESS *REACH_WF_INDUCT*

$$P, J : \frac{q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\forall M :: p \wedge (m = M) \rightarrowtail (p \wedge (m \prec M)) \vee q)}{p \rightarrowtail q}$$

Theorem 5.3.2 $\rightsquigarrow$ BOUNDED PROGRESS *CON_WF_INDUCT*

$$P, J : \frac{(q \rightsquigarrow q) \ \wedge \ (\forall M :: p \wedge (m = M) \rightsquigarrow (p \wedge (m \prec M)) \vee q)}{p \rightsquigarrow q}$$

Note: The notation used above is overloaded. Without overloading, $p \wedge (m = M) \rightsquigarrow (p \wedge (m \prec M)) \vee q$ can be written as:

$$(\lambda s.\ p.s \wedge (m.s = M)) \ \rightsquigarrow \ (\lambda s.\ (p.s \wedge (m.s \prec M)) \vee q.s)$$

Figure 5.5: *Bounded progress principle for progress and convergence.*

Theorem 5.3.3 ROUND DECOMPOSITION *CON_BY_sWF_i*
For all finite and non-empty sets A and all well-founded relations $\prec \in A \times A$:

$$P : \frac{(\circlearrowright J) \ \wedge \ (\forall n : n \in A : J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \rightsquigarrow q.n)}{J \vdash \mathsf{true} \rightsquigarrow (\forall n : n \in A : q.n)}$$

Proof:

$J \vdash \mathsf{true} \rightsquigarrow (\forall n : n \in A : q.n)$

$\Leftarrow$ { $\rightsquigarrow$ CONJUNCTION$_{91}$ }

$(\forall n : n \in A : J \vdash \mathsf{true} \rightsquigarrow q.n)$

$\Leftarrow$ { WELL-FOUNDED INDUCTION$_{50}$ }

$(\forall n : n \in A : (\forall m : m \prec n : J \vdash \mathsf{true} \rightsquigarrow q.m) \ \Rightarrow \ (J \vdash \mathsf{true} \rightsquigarrow q.n))$

If n is a minimal element then:

$(\forall m : m \prec n : J \vdash \mathsf{true} \rightsquigarrow q.m) \ \Rightarrow \ (J \vdash \mathsf{true} \rightsquigarrow q.n)$

$\Leftarrow$ { predicate calculus }

$J \vdash \mathsf{true} \rightsquigarrow q.n$

$=$ { n is a minimal element, hence there is no m such that $m \prec n$ }

$J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \rightsquigarrow q.n$

If n is not a minimal element then:

$(\forall m : m \prec n : J \vdash \mathsf{true} \rightsquigarrow q.m) \ \Rightarrow \ (J \vdash \mathsf{true} \rightsquigarrow q.n)$

```
prog     doMin
read     $\{x.n \mid n \text{ in the tree}\} \cup \{y.n \mid n \text{ in the tree}\}$
write    $\{y.n \mid n \text{ in the tree}\}$
init     true
assign   $y.n := \mathsf{min}\{y.m \mid m \text{ is a son of } n\} \ \mathsf{min}\ x.n$
```

◀

Figure 5.6: *Computing minimum inputs.*

$\Leftarrow$ $\{ \rightsquigarrow$ CONJUNCTION$_{91}$ $\}$

$(J \vdash \mathsf{true} \rightsquigarrow (\forall m : m \prec n : q.m)) \ \Rightarrow \ (J \vdash \mathsf{true} \rightsquigarrow q.n)$

$\Leftarrow$ $\{ \rightsquigarrow$ TRANSITIVITY$_{91}$ $\}$

$J \vdash (\forall m : m \prec n : q.m) \rightsquigarrow q.n$

$\Leftarrow$ $\{ \rightsquigarrow$ STABLE SHIFT$_{91}$ $\}$

$(\forall m : m \prec n : q.m) \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\vdash \circlearrowleft J) \ \wedge \ (J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \rightsquigarrow q.n)$

$\Leftarrow$ $\{ \rightsquigarrow$ CONFINEMENT$_{91}$, confinement is preserved by $\forall$ $\}$

$(\vdash \circlearrowleft J) \ \wedge \ (\forall n : n \in A : (J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \rightsquigarrow q.n))$

▲

Theorem 5.3.3 above is called ROUND DECOMPOSITION for the following reason. One can view a loop in terms of rounds. Each iteration step makes the system advance to the next round, until the final round is reached. In a sequential system the rounds are totally ordered. This does not have to be the case in a distributed system, although well-foundedness will still be required. In sequential programming, invariants are used to specify the obligation of each iteration step. Imagine a distributed system that iterates along a (finite and well-founded) ordering $\prec$ over the domain of rounds to establish $(\forall n : n \in A : q_n)$. Theorem 5.3.3 states that it suffices to have:

$$J \wedge (\forall m : m \prec n : q_m) \vdash \mathsf{true} \rightsquigarrow q.n$$

for each round n. The above specifies the obligation of each round. The predicate $J \wedge (\forall m : m \prec n : q_m)$ can be viewed as some sort of loop invariant.

5.3.1 An Example: Computing Minimum

Imagine again a tree of processes with the root α. Each process n has a variable $x.n$. The values of the $x.n$'s are set up by the environment and the processes themselves cannot influence them. The task is to compute the minimum value of the $x.n$'s and to make this value known to the root α. The computation has to be self-stabilizing such that if the environment issues a new value for some $x.n$, a new minimum will be automatically re-computed. Actually this is not a difficult requirement. A standard distributed minimum computation, repeated forever, will do. The algorithm is presented in Figure 5.6.

Still, there are other aspects besides the clarity of the solution that we wish to bring into light. First, we want to show how the requirement for self-stabilization can be expressed in terms of $\rightsquigarrow$. Second, we are interested in the calculation, in particular in how informal ideas and strategies are translated to the formal level. In addition, to make the problem more interesting, we will generalize it by considering the greatest lower bound operator $\sqcap$ of an arbitrary semi-lattice[4] $\prec$ instead of the standard min operator.

Let us now be more precise with our tree. The tree is finite and A is the set of all processes —also called nodes— in the tree. The set of all *sons* of a node n is given by $S.n$. If we define $S^0.n = \{n\}$ and $S^{i+1} = S \circ S^i$, we can define the 'transitive' closure of S, denoted by S^+, as $S^+.n = \cup\{S^i.n \mid 0 < i\}$ and the 'transitive and reflexive' closure of S, denoted by S^*, as $S^*.n = S^+.n \cup S^0.n$. The set $S^*.n$ corresponds with the set of *descendants* of n and $S^+.n$ with the set of *proper descendants* of n. We can regard S, S^+, and S^* as relations. For example: $m\ S\ n = m \in S.n$. Notice that as a relation S^+ is well-founded.

For the sake of readability, confinement constraints will be omitted from our calculation. When a mechanical verification is attempted, one should however be prepared to deal with every detail explicitly.

Before we continue, let us first introduce a function *map* which will be useful in specifying the problem. For any function f and any set V, let $f * V$ (the map[5] of f on V) be defined as:

$$f * V \;=\; \{f.x \mid x \in V\} \tag{5.3.1}$$

Map satisfies the following properties:

$$f * (g * V) \;=\; (f \circ g) * V \tag{5.3.2}$$
$$f * (V \cup W) \;=\; (f * V) \cup (f * W) \tag{5.3.3}$$
$$f * (\cup V) \;=\; \cup((f*) * V) \tag{5.3.4}$$

Using the map operator the problem of minimum computation can be stated as: compute $\sqcap(x * S^*.\alpha)$. If we let the result to be stored in $y.\alpha$, the problem can be specified as follows:

$$\mathsf{M1}:\quad (s = S)\ {}_{\mathsf{doMin}}\!\vdash \mathsf{true} \rightsquigarrow (y.\alpha = \sqcap(x * S^*.\alpha))$$

Let us first introduce some abbreviations which we will use later. For all $n \in A$:

$$\mathsf{Ok}^n.S \;=\; (y.n = \sqcap(x * S^*.n)) \tag{5.3.5}$$
$$\mathsf{preOk}^n.S \;=\; (\forall m : m \in S^+.n : \mathsf{Ok}^m) \tag{5.3.6}$$

[4] Another approach would be to use an idempotent, commutative, and associative operator $\oplus$ (also called a *regular algebra*). A nice treatment was given by Backhouse, Eijnde, and van Gasteren in [BEvG94].

[5] The notation is borrowed from Functional Programming

Ok^n states that process n has a *'correct'* value of $y.n$, which here means that the value of $y.n$ is equal to the minimum of all x's of the descendants of n. preOk^n states that all processes that 'precede' n, which here means being proper descendants of n, have correct values of their y's.

As a preparation for further calculation let us express $\mathsf{M1}$ in terms of Ok and then strengthen the goal to include a similar goal for each process n:

$$\begin{aligned}
&\quad \mathsf{true} \rightsquigarrow (y.\alpha = \sqcap(x * S^*.\alpha)) \\
= &\qquad \{ \text{ definition } \mathsf{Ok} \ \} \\
&\quad \mathsf{true} \rightsquigarrow \mathsf{Ok}^\alpha.S \\
\Leftarrow &\qquad \{ \rightsquigarrow \textsc{Substitution}_{91} \ \} \\
&\quad \mathsf{true} \rightsquigarrow (\forall n : n \in A : \mathsf{Ok}^n.S)
\end{aligned}$$

So, $\mathsf{M1}$ can be refined by $\mathsf{M2}$:

$$\mathsf{M2}: \quad (s = S) \ {}_P\!\vdash \mathsf{true} \rightsquigarrow (\forall n : n \in A : \mathsf{Ok}^n.S)$$

To establish Ok our strategy is as follows. Suppose that somehow we can establish Ok^m for all proper descendants m of n, then we might try to establish Ok^n using this knowledge. This is done repeatedly until Ok^α is established. This sounds very much like round decomposition: A is the set of rounds, ordered by S^+, and Ok^n is the goal of round n. The following calculation will make this apparent:

$$\begin{aligned}
&\quad (s = S) \vdash \mathsf{true} \rightsquigarrow (\forall n : n \in A : \mathsf{Ok}^n.S) \\
\Leftarrow &\qquad \{ \ S^+ \text{ is well founded; } \textsc{Round Decomposition}_{93} \ \} \\
&\quad (\forall n : n \in A : (s = S) \ \wedge \ (\forall m : m \in S^+.n : \mathsf{Ok}^m.S) \vdash \mathsf{true} \rightsquigarrow \mathsf{Ok}^n.S) \\
= &\qquad \{ \text{ definition of } \mathsf{preOk} \ \} \\
&\quad (\forall n : n \in V : (s = S) \wedge \mathsf{preOk}^n.S \vdash \mathsf{true} \rightsquigarrow \mathsf{Ok}^n.S)
\end{aligned}$$

Notice how the final specification reflects our strategy (establish Ok^n given that all proper descendants are Ok).

Furthermore, we observe that the task of establishing Ok^n can be delegated to process n, which we will call $\mathsf{doMin}.n$. If we insist that each process n only writes to $y.n$ then these processes are write-disjoint, which is nice because we can now apply the $\textsc{Transparency}_{92}$ principle. We continue the calculation:

$$\begin{aligned}
&\quad (s = S) \wedge \mathsf{preOk}^n.S \ {}_{\mathsf{doMin}}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{Ok}^n.S \\
\Leftarrow &\qquad \{ \ \mathsf{doMin} = ([\![n : n \in A : \mathsf{doMin}.n), \rightsquigarrow \textsc{Transparency}_{92} \ \} \\
&\quad ({}_{\mathsf{doMin}}\!\vdash \circlearrowright((s = S) \wedge \mathsf{preOk}^n.S)) \ \wedge \ ((s = S) \wedge \mathsf{preOk}^n.S \ {}_{\mathsf{doMin}.n}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{Ok}^n.S) \\
\Leftarrow &\qquad \{ \rightsquigarrow \textsc{Introduction}_{91}, \circlearrowright \textsc{Compositionality}_{60} \ \} \\
&\quad ({}_{\mathsf{doMin}}\!\vdash \circlearrowright((s = S) \wedge \mathsf{preOk}^n.S)) \ \wedge \ ({}_{\mathsf{doMin}.n}\!\vdash \circlearrowright((s = S) \wedge \mathsf{preOk}^n.S \wedge \mathsf{Ok}^n.S)) \ \wedge \\
&\quad ({}_{\mathsf{doMin}.n}\!\vdash (s = S) \wedge \mathsf{preOk}^n.S \ \mathsf{ensures} \ \mathsf{Ok}^n.S)
\end{aligned}$$

To summarize, we have refined $\mathsf{M2}$ to the following specification:

Let $\mathsf{doMin} = ([\![n : n \in A : \mathsf{doMin}.n)$ such that $\mathbf{w}(\mathsf{doMin}.n) = \{y.n\})$. For all $n \in V$:

$$\begin{array}{lll} \mathsf{M3.a}: & {}_{\mathsf{doMin}}\vdash & \circlearrowright((s = S) \wedge \mathsf{preOk}^n.S) \\ \mathsf{M3.b}: & {}_{\mathsf{doMin}.n}\vdash & \circlearrowright((s = S) \wedge \mathsf{preOk}^n.S \wedge \mathsf{Ok}^n.S) \\ \mathsf{M3.c}: & {}_{\mathsf{doMin}.n}\vdash & (s = S) \wedge \mathsf{preOk}^n.S \text{ ensures } \mathsf{Ok}^n.S \end{array}$$

---◀

In particular, $\mathsf{M3.c}$ states that we have to establish Ok^n from preOk^n. This can be done by computing $\sqcap(x * S^*.n)$ from $\sqcap(x * S^*.m)$ of all sons m of n:

$$\sqcap(x * S^*.n) \;=\; (\sqcap((\sqcap \circ (x*) \circ S^*) * S.n)) \sqcap x.n \tag{5.3.7}$$

Proof: To prove the above we use the following property of a semi-lattice. In a semi-lattice, the corresponding $\sqcap$ operator satisfies:

$$\sqcap(\cup V) \;=\; \sqcap(\sqcap * V) \tag{5.3.8}$$

An instance of the above is: $\sqcap(U \cup V) = (\sqcap U) \sqcap (\sqcap V)$. Now let us prove the lemma above:

$$\begin{array}{ll} & \sqcap(x * S^*.n) \\ = & \quad \{ \text{ a property of } S^* \ \} \\ & \sqcap(x * ((\cup\{m : m \in S.n : S^*.m\}) \cup \{n\})) \\ = & \quad \{ \text{ definition } * \ \} \\ & \sqcap(x * ((\cup(S^* * S.n)) \cup \{n\})) \\ = & \quad \{ \text{ properties of } *\text{: (5.3.4), (5.3.3), and (5.3.2) } \} \\ & \sqcap((\cup(((x*) \circ S^*) * S.n)) \cup \{x.n\}) \\ = & \quad \{ \text{ (5.3.8) and (5.3.2) } \} \\ & (\sqcap((\sqcap \circ (x*) \circ S^*) * S.n)) \sqcap x.n \end{array}$$

▲

The lemma suggests that $\mathsf{Ok}^n.S$ can be established by the assignment:

$$y.n := (\sqcap((\sqcap \circ (x*) \circ S^*) * S.n)) \sqcap x.n$$

However, $\mathsf{preOk}.S$ implies that for all sons m of n, $y.m = \sqcap(x * S^*.m)$. It follows that the expression $(\sqcap \circ (x*) \circ S^*) * S.n$ in the assignment above can be replaced by $y * S.n$. So, the assignment $y.n := (\sqcap(y * S.n)) \sqcap x.n$ will do the job, and since $s = S$, we can replace any reference to the constant S with a reference to the variable s.

A program that meets requirement $\mathsf{M3}$ is given in Figure 5.7. The program is essentially similar as the first program given in Figure 5.6. Notice that the topology of the tree is an input variable to the program (s). Hence, by $\mathsf{M1}$ the program is capable to recover from topology changes by the environment, as long as s defines a finite tree over A with root α. The reader may notice that the program $\mathsf{doMin}.n$ is allowed to read all $y.m$'s while it actually only reads from the $y.m$'s of the sons of n. This redundancy is necessary to accommodate topology changes (n may be coupled to an entirely different set of sons).

doMin $= ([\!]\, n : n \in A : \text{doMin}.n)$ where doMin.n is:

```
prog     doMin.$n$
read     $\{y.m \mid m \in A\} \cup \{s.n, x.n, y.n\}$
write    $\{y.n\}$
init     $(A, s)$ is a finite tree with root $\alpha$
assign   $y.n := (\sqcap(y * s.n)) \sqcap x.n$
```

Figure 5.7: *A general algorithm to compute minimum.*

5.4 Adding Asynchronicity

Any UNITY program can be directly implemented in a machine with a shared memory system. However, in many distributed systems, processes have to communicate through asynchronous channels. Unfortunately, adding asynchrony does not always preserve properties of a program. Usually, self-stabilization in an asynchronous system is dealt with as a separate problem. Dolev, Israeli, and Moran were the first to address the issue [DIM90]. Until now, no theorem describing a unified condition under which addition of asynchrony preserves convergence exists. What one still can do is to parameterize a proof with a variable degree of asynchrony. We will show an example of such a proof in Section 5.5.

In their paper [DIM90] Dolev, Israeli, and Moran considered a system with registers, called *link-registers*, inserted between any pair of processes. A link-register is written by one process, and read by the other, but no two processes may write to the same register. Dolev, Israeli, and Moran introduced a *read/write daemon* to enforce this rule, but essentially their system is just a write-disjoint system. For example, by adding link-registers, the minimum computing program in Figure 5.7 becomes:

```
prog     doMin.$n$
read     $\{r.l.m \mid l, m \in A\} \cup \{f.n, s.n, x.n, y.n\}$
write    $\{y.n\} \cup \{r.m.n \mid m \in A\}$
init     $(A, s)$ is a finite tree with root $\alpha$
assign   $y.n := (\sqcap(r.n * s.n)) \sqcap x.n$
[]       $r.(f.n).n := y.n$
```

where $f.n$ stores the identity of the father of n, and $r.m.n$ is the link-register between processes m and n. It is intended to be m's view to $y.n$.

There are several ways to model asynchronous channels, for example by using queues: a reader adds messages to the front of a queue, and a receiver retrieves from the back. One can also use separate write and read histories. Another model using write histories and read-counters was shown in Chapter 2 (this model of channels is write-disjoint). In UNITY link-registers also, in a sense, abstract from unreliable channels. A datum in a link-register can be overwritten by the sender as the sender cannot force the reader to immediately read the datum it has just written. So, data can be lost. If a program is proven to work correctly using link-registers, it will therefore also works correctly using perfect channels.

The first problem in adding asynchrony has to do with atomicity. Consider the following programs $P[\!]Q$ and $P'[\!]Q'$. The latter is obtained by adding a link-register r to the first.

```
prog    $P$                                   prog    $P'$
assign  $x := f.x$                            assign  $x := f.x$
                                              []      $r := x$
prog    $Q$
assign  if $b$ then $y, b := x$, false        prog    $Q'$
[]      if $\neg b$ then $b :=$ true          assign  if $b$ then $y, b := r$, true
                                              []      if $\neg b$ then $b :=$ false
```

In $P[\!]Q$, if a datum is being transferred from x to y, b will simultaneously set to false. In particular, the following property holds for $P[\!]Q$:

$$(\forall X :: \ \vdash b \wedge (x = X) \ \textsf{unless} \ (\neg b \wedge (y = x)) \vee (x = f.X))$$

This property is however not satisfied by $P'[\!]Q'$, because the register may still contain an old datum, and assigning the value to y will not establish $\neg b \wedge (y = x)$. In $P[\!]Q$, $\neg b \wedge (y = x)$ can be established by a single atomic action. Introducing r reduces the atomicity, and the problem described above arises.

The second problem is caused by 'bad' registers or channel values which travel around the system forever, preventing it from stabilization. Consider the following programs P and Q:

$$P : a := b \quad \text{and} \quad Q : b := a$$

The program $P[\!]Q$ satisfies:

$$\textsf{true} \vdash \ \textsf{true} \rightarrowtail (a = b) \tag{5.4.1}$$

$$\vdash \ \circlearrowleft (a = b) \tag{5.4.2}$$

Let us now add link-registers to $P[\!]Q$: $r.a$ is intended to be Q's view of a and $r.b$ is P's view of b. We obtain a program R with the following actions:

$$\begin{array}{lll} (a_1) & a := r.b & [\!] \\ (a_2) & r.a := a & [\!] \\ (a_3) & b := r.a & [\!] \\ (a_4) & r.b := b & \end{array}$$

The transformation does not preserve (5.4.1). Consider the execution:

$$(a_1; a_3; a_2; a_4)^*$$

The execution is fair, but in this execution the value of $r.a$ and $r.b$ will simply rotate around. Consequently, if initially $r.a \neq r.b$, then $a = b$ may never happen. The transformation does not preserve (5.4.2) either, since the described stability can be destroyed by a_1 or a_2.

```
prog    MinDist
read    $\{d.a.b \mid a,b \in V\}$
write   $\{d.a.b \mid a,b \in V\}$
init    true
assign  $([\!]a : a \in V : d.a.a := 0)$
$[\!]$       $([\!]a,b : a,b \in V \wedge a \neq b : d.a.b := \min\{d.a.b' + 1 \mid b' \in N.b\})$
```

Figure 5.8: *A self-stabilizing minimum distance algorithm in UNITY.*

5.5 Lentfert's FSA Algorithm

Recall that one of the goals of this thesis is to present a (mechanical) verification of Lenfert's work in [Len93] on a general, self-stabilizing, and distributed algorithm — what here is called Fair and Successive Approximation (FSA) algorithm— to compute, for example, the minimal distance between all pairs of nodes in a network. Lentfert's thesis included a detailed formal proof of the algorithm. For several reasons, we will re-present the proof. First, we want to specify and reason about the algorithm in terms of convergence rather than in terms of leads-to-a-stable or eventually-implies used in [Len93]. The resulting proof is more concise and direct than the one in [Len93]. A concise proof is also easier to verify mechanically. Second, the decision about the choice of communication protocol will be postponed as long as possible. It will be highlighted which results do not depend on the choice of communication model. Third, the proof will show a quite clear separation between the programming and the more intrinsic aspects of the problem.

An instance of the FSA algorithm was shown in Figure 2.2. The UNITY version of the algorithm is given in Figure 5.8. The program computes the minimal distance between any pair of nodes in the network (V, N). Here, V is the set of all nodes in the network, and N is a function such that $N.a$ is the set of all neighbors of a. The network (V, N) will be assumed to be *finite*. For the sake of simplicity we will also assumed it to be *bi-directional*. We will start from this program. On the way, we will generalize the results to obtain the FSA algorithm.

Note that the initial condition of the program is true, which suggests that the program may be self-stabilizing. We will show that it is. The specification of MinDist can be stated as follows:

$$\mathsf{MD0}: \quad \mathsf{true}\ {}_{\mathsf{MinDist}}\!\vdash \mathsf{true} \rightsquigarrow (\forall a,b : a,b \in V : d.a.b = \mathsf{Dist}.a.b) \tag{5.5.1}$$

where $\mathsf{Dist}.a.b$ denotes the actual minimal distance between a and b[6].

[6] We carelessly call $\mathsf{Dist}.a.b$ the minimal distance *between* a and b, or sometimes also *from* a *to* b. The network is assumed to be bi-directional, meaning that $b \in N.c = c \in N.b$. For the simple notion of minimal distance as defined by (5.5.2) and (5.5.3) the above two interpretations are indeed interchangeable. One may want to parameterize the network with a cost function, which specifies the cost, or 'distance' of going through a link. In the simple notion of distance this cost is always 1. If a

It is known that the function Dist is characterized by the following equations:

$$(\forall a : a \in V : \mathsf{Dist}.a.a = 0) \tag{5.5.2}$$

$$(\forall a, b : a, b \in V \wedge a \neq b : \ \mathsf{Dist}.a.b = \mathsf{min}\{\mathsf{Dist}.a.b' + 1 \mid b' \in N.b\}) \tag{5.5.3}$$

One may remark that upon reaching its fix point, d in the program Mindist in Figure 5.8 will satisfy the above equation, and hence $d = \mathsf{Dist}$. However, this does not help us much since we still do not know whether the program will converge to its fix-point from *any given state*. In fact, here lies the hardest part of the problem.

The program MinDist can be implemented as a distributed program by associating a process for each node b. Each process b maintains the minimum distance between b and any other node a, which is stored in a variable $d.a.b$. Notice that Mindist assumes that a process b can read the variable $d.a.b'$ owned by a neighbor b'. Many distributed systems require however a higher degree of asynchrony. For example, it may be required that the processes communicate through channels. We also want to take this aspect into account. So, we are going to design a slightly different MinDist. For example, if we use link-registers to model channels, then the program MinDist will look like:

```
prog     MinDist
init     true
assign   $([\!]a : a \in V : d.a.a := 0)$
[]       $([\!]a, b : a, b \in V \wedge a \neq b : d.a.b := \mathsf{min}\{r.a.b.b' + 1 \mid b' \in N.b\})$
[]       $([\!]a, b, c : a, b, c \in V \wedge b \in N.c : r.a.c.b := d.a.b)$
```

where $r.a.c.b$ is intended to be c's view to $d.a.b$. However, for now we do not want to concern ourselves with the specific model and implementation of the channels. It suffices to know that they exist.

First of all, we observe from equations (5.5.2) and (5.5.3) that $\mathsf{Dist}.a$ can be computed without any information about $\mathsf{Dist}.a'$, given $a' \neq a$. So, we can delegate the task of maintaining $d.a$ to a component program (which may consists of more components). Let us call this component $\mathsf{MinDist}.a$. If we also insists that these components are pair-wise write-disjunct then we can use the TRANSPARENCY law to delegate the above mentioned task. More precisely, by applying the $\rightsquigarrow$ WRITE-DISJOINT CONJUNCTION$_{92}$ —which is a corollary of the TRANSPARENCY law—, we can refine MD0 to MD1 below. For all $a, b \in V$:

$$\text{MD1.a}: \quad \mathsf{true} \ {}_{\mathsf{MinDist}.a}\!\vdash \mathsf{true} \rightsquigarrow (\forall b : b \in V : d.a.b = \mathsf{Dist}.a.b) \tag{5.5.4}$$

$$\text{MD1.b}: \quad \mathsf{MinDist} \ = \ ([\!]a' : a' \in V : \mathsf{MinDist}.a') \tag{5.5.5}$$

$$\text{MD1.c}: \quad (a \neq b) \Rightarrow (\mathsf{MinDist}.a \div \mathsf{MinDist}.b) \tag{5.5.6}$$

We will now divide the execution of each $\mathsf{MinDist}.a$ into imaginary phases or rounds. The rounds are taken from $\mathbb{N}$, ordered by $<$. Each round has its obligation such that

cost function is used, it may happen that the cost of the link from b to c differs from the cost of going from c to b. So, despite the bi-directionality, distance has direction. Whether we should interpret $\mathsf{Dist}.a.b$ as the 'distance' from a to b or from b to a depends on whether one interprets $a \in N.b$ as '*there exists a link from a to b*' or the other way around.

when all rounds are passed, $\mathsf{MinDist}.a$ has converged to a situation where $d.a = \mathsf{Dist}.a$, and hence $\mathsf{MD1.a}$ is fulfilled. A quite obvious choice is to require that at the end of round n, all processes b with the actual distance n from a have converged to a situation where $d.a.b = \mathsf{Dist}.a.b$. Since convergence is conjunctive it follows that at the end of of round n, all processes b with $\mathsf{Dist}.a.b \leq n$ have converged to $d.a.b = \mathsf{Dist}.a.b$. Let n_{max} be the greatest minimal distance across the network (V, N) —it is also called the *diameter* of the network. It follows that at round n_{max} *all* processes b have converged to $d.a.b = \mathsf{Dist}.a.b$. Recall the ROUND DECOMPOSITION$_{93}$ principle from Section 5.3. As we will see soon, the strategy above is actually an instance of this principle. However, let us generalize the strategy by taking a finite set A instead of $\mathbb{N}$ as the domain of the rounds, ordered by a well-founded relation $\prec$.

In addition to stabilizing the value of d, the obligation of a round must also cover the stabilization of the channels. One may suspect that the channels initially contain 'bad' values which may keep circulating in the system, preventing it from stabilizing. We have to show that this is not the case.

Let us first introduce a few abbreviations regarding the obligation of a round. Let A be a finite and non-empty set of rounds ordered by $\prec$ which is well-founded. For all $n \in A$, $b \in V$:

$$\begin{aligned}
\mathsf{ok}^n_b.X &= X \textit{ is an acceptable value for node } b \textit{ at round } n && (5.5.7)\\
\mathsf{Ok}^n_b &= \mathsf{ok}^n_b.(d.a.b) && (5.5.8)\\
\mathsf{dataOk}^n &= (\forall b : b \in V : \mathsf{Ok}^n_b) && (5.5.9)\\
\mathsf{cOk}^n_b &= \textit{the communication obligation of process } b \textit{ for round } n && (5.5.10)\\
\mathsf{comOk}^n &= (\forall b : b \in V : \mathsf{cOk}^n_b) && (5.5.11)\\
\mathsf{preOk}^n &= (\forall m : m \prec n : \mathsf{dataOk}^m \wedge \mathsf{comOk}^m) && (5.5.12)
\end{aligned}$$

We intentionally leave the meaning of '*acceptable value*' and '*communication obligation*' in the definition of ok and cOk unspecified because first we want to expose a general structure of $\mathsf{MinDist}$ —or in general, of the FSA algorithm— which is derivable without detailed information of what is actually being computed. For now, the only thing we know is that:

$$\mathsf{dataOk}^\eta \Rightarrow (\forall b : b \in V : d.a.b = \mathsf{Dist}.a.b) \qquad (5.5.13)$$

for some $\eta \in A$. This η is a generalization of n_{max}, the diameter of the network, mentioned earlier.

Now let us do some calculation to refine $\mathsf{MD1.a}$. For the sake of readability, confinement constraints will be omitted from formulas.

$$\begin{aligned}
& \mathsf{true} \vdash \mathsf{true} \rightsquigarrow (\forall b : b \in V : d.a.b = \mathsf{Dist}.a.b)\\
\Leftarrow \quad & \{ \rightsquigarrow \text{SUBSTITUTION}_{91}, (5.5.13) \}\\
& \mathsf{true} \vdash \mathsf{true} \rightsquigarrow \mathsf{dataOk}^\eta\\
\Leftarrow \quad & \{ (\dagger)\ \eta \in A, \rightsquigarrow \text{SUBSTITUTION}_{91} \}
\end{aligned}$$

$$
\begin{array}{ll}
& \mathsf{true} \vdash \mathsf{true} \rightsquigarrow (\forall n : n \in A : \mathsf{dataOk}^n \wedge \mathsf{comOk}^n) \\
\Leftarrow & \{\ (\ddagger)\ \textsc{Round Decomposition}_{93}, \text{Definition of } \mathsf{preOk}\ \} \\
& (\forall n : n \in A : \mathsf{preOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{dataOk}^n \wedge \mathsf{comOk}^n) \\
\Leftarrow & \{\ \textsc{Accumulation}_{91}\ \} \\
& (\forall n : n \in A : (\mathsf{preOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{dataOk}^n) \ \wedge\ (\mathsf{preOk}^n \vdash \mathsf{dataOk}^n \rightsquigarrow \mathsf{comOk}^n)) \\
\Leftarrow & \{\ \rightsquigarrow \textsc{Stable Shift}_{91}\ \} \\
& (\forall n : n \in A : (\mathsf{preOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{dataOk}^n) \ \wedge\ (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{comOk}^n)) \\
\Leftarrow & \{\ \text{definition of } \mathsf{dataOk} \text{ and } \mathsf{comOk},\ \rightsquigarrow \textsc{Conjunction}_{91}\ \} \\
& (\forall n, b : n \in A \wedge b \in V : (\mathsf{preOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{Ok}^n_b) \ \wedge\ (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{cOk}^n_b))
\end{array}
$$

The calculation implies that in order to meet MD1.a it suffices for MinDist.a, at each round n, to establish Ok^n_b (the computation obligation of node b for round n) and cOk^n_b (the communication obligations of node b for round n), given that the obligations of all preceding rounds have been fulfilled. This formalizes the strategy described a few paragraphs earlier. Notice the application of the Round Decomposition principle (‡). Notice also how the communication obligation was taken into account (†). So, based on the calculation we can refine MD1.a to MD2 defined as follows. For all $a, b \in V$ and $n \in A$:

$$\text{MD2.a :} \quad \mathsf{preOk}^n \ {}_{\mathsf{MinDist}.a}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{Ok}^n_b \tag{5.5.14}$$
$$\text{MD2.b :} \quad \mathsf{preOk}^n \wedge \mathsf{dataOk}^n \ {}_{\mathsf{MinDist}.a}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{cOk}^n_b \tag{5.5.15}$$

Just as we have split MinDist into MinDist.a, we are now going to split each MinDist.a into smaller programs. It seems reasonable to delegate the task of establishing Ok^n_b and cOk^n_b in the specifications above to a component b. Let us call this component MinDist.$a.b$. Using the $\rightsquigarrow$ Transparency$_{92}$ law we can refine MD2 to MD3 defined as follows. For all $a, b, c \in V$ and $n \in A$:

$$\text{MD3.a :} \quad \mathsf{preOk}^n \ {}_{\mathsf{MinDist}.a.b}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{Ok}^n_b \tag{5.5.16}$$
$$\text{MD3.b :} \quad \mathsf{preOk}^n \wedge \mathsf{dataOk}^n \ {}_{\mathsf{MinDist}.a.b}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{cOk}^n_b \tag{5.5.17}$$
$$\text{MD3.c :} \quad \mathsf{MinDist}.a \ = \ ([\![a' : a' \in V : \mathsf{MinDist}.a.a') \tag{5.5.18}$$
$$\text{MD3.d :} \quad (b \neq c) \ \Rightarrow \ (\mathsf{MinDist}.a.b \div \mathsf{MinDist}.a.c) \tag{5.5.19}$$

The decision to divide MinDist into write-disjoint components is a design decision. One can opt not to do this, but then of course one cannot benefit from the Transparency law. Note that MD3.a suggests that MinDist.$a.b$ writes to, at least, $d.a.b$.

Let us now add some more details to cOk. In establishing Ok^n_b in MD3.a we know that preOk^n holds, which means that Ok^m_c holds for any node c —in particular if c is a neighbor of b— and round $m \prec n$. The computation required to establish Ok^n_b will have to, one way or another, exploit this fact. From its definition in (5.5.8) it seems that Ok^m_c is a predicate that only concerns $d.a.c$. If the communication between processes has to take place through channels, then the component $\mathsf{MinDist}.a.b$ cannot read from $d.a.c$ (for $b \neq c$). Therefore we introduce a new variable r, and decide that each $\mathsf{MinDist}.a.b$ should maintain $r.a.b.b'$ as a copy of $d.a.b'$ for each neighbor b' of b. The obligation of the communication subsystem is simply to guarantee that the r's eventually contain the copies of the d's. This suggests the following specification for cOk:

$$\mathsf{cOk}_b^n \;\Rightarrow\; (\forall c : b \in N.c : r.a.c.b = d.a.b)$$

Logical as it may seem, the above specification is too strong. In $\mathsf{MD3.b}$ it is required not only to establish cOk_b^n, but also to converge to it. Roughly, this implies that $r.a.c.b = d.a.b$ must be held stable, which is not reasonable as $d.a.b$ is likely to change during an execution and we may not want to update $r.a.c.b$ immediately after each change of $d.a.b$. However, as will be made apparent in the next calculation, all that we need is that $r.a.c.b$ contains an acceptable value for round n. Whether or not it contains the most recent copy of $d.a.b$ is less important.

Below is a partial definition of cOk. For any $a, b \in V$ and $n \in A$:

$$\mathsf{cOk}_b^n \;\Rightarrow\; (\forall c : b \in N.c : \mathsf{ok}_b^n.(r.a.c.b)) \tag{5.5.20}$$

Now let us calculate for $\mathsf{MD3.a}$:

$$
\begin{array}{ll}
 & \mathsf{preOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{Ok}_b^n \\
\Leftarrow & \quad \{ \rightsquigarrow \textsc{Introduction}_{91} \} \\
 & (\vdash \circlearrowleft \mathsf{preOk}^n) \,\wedge\, (\vdash \circlearrowleft (\mathsf{preOk}^n \wedge \mathsf{Ok}_b^n)) \,\wedge\, (\vdash \mathsf{preOk}\ \mathsf{ensures}\ \mathsf{Ok}_b^n) \\
\Leftarrow & \quad \{ \text{definition}_{42} \text{ of } \mathsf{ensures} \} \\
 & (\vdash \circlearrowleft \mathsf{preOk}^n) \,\wedge\, (\vdash \circlearrowleft (\mathsf{preOk}^n \wedge \mathsf{Ok}_b^n)) \,\wedge\, (\vdash \mathsf{preOk}^n\ \mathsf{unless}\ \mathsf{Ok}_b^n) \,\wedge \\
 & (\exists a : a \in \mathbf{a}(\mathsf{MinDist}.a.b) : \{\mathsf{preOk}^n\}\ a\ \{\mathsf{Ok}_b^n\}) \\
\Leftarrow & \quad \{ \mathsf{unless}\ \textsc{Post-weakening}_{46} \} \\
 & (\vdash \circlearrowleft \mathsf{preOk}^n) \,\wedge\, (\vdash \circlearrowleft (\mathsf{preOk}^n \wedge \mathsf{Ok}_b^n)) \,\wedge\, (\vdash \mathsf{preOk}^n\ \mathsf{unless}\ \mathsf{false}) \,\wedge \\
 & (\exists a : a \in \mathbf{a}(\mathsf{MinDist}.a.b) : \{\mathsf{preOk}^n\}\ a\ \{\mathsf{Ok}_b^n\}) \\
\Leftarrow & \quad \{ \text{definition}_{44} \text{ of } \circlearrowleft \} \\
 & (\vdash \circlearrowleft \mathsf{preOk}^n) \,\wedge\, (\vdash \circlearrowleft (\mathsf{preOk}^n \wedge \mathsf{Ok}_b^n)) \,\wedge \\
 & (\exists a : a \in \mathbf{a}(\mathsf{MinDist}.a.b) : \{\mathsf{preOk}^n\}\ a\ \{\mathsf{Ok}_b^n\})
\end{array}
$$

For the single Hoare triple in the last formula we continue our calculation:

$$
\begin{array}{ll}
 & \{\mathsf{preOk}^n\}\ a\ \{\mathsf{Ok}_b^n\} \\
\Leftarrow & \quad \{ \text{definition (5.5.12) of } \mathsf{preOk}\text{, pre-condition strengthening} \} \\
 & \{(\forall m : m \prec n : \mathsf{comOk}^m)\}\ a\ \{\mathsf{Ok}_b^n\} \\
\Leftarrow & \quad \{ \text{definition (5.5.11) of } \mathsf{comOk}\text{, pre-condition strengthening} \} \\
 & \{(\forall m, b' : m \prec n \wedge b' \in N.b : \mathsf{cOk}_{b'}^m)\}\ a\ \{\mathsf{Ok}_b^n\} \\
\Leftarrow & \quad \{ \text{definition (5.5.20) of } \mathsf{cOk} \text{ and (5.5.8), pre-condition strengthening} \} \\
 & \{(\forall m, b' : m \prec n \wedge b' \in N.b : \mathsf{ok}_{b'}^m.(r.a.b.b'))\}\ a\ \{\mathsf{ok}_b^n.(d.a.b)\}
\end{array}
$$

$$\begin{array}{ll} \Leftarrow & \{\ (\dagger)\ \text{choose } a \text{ and introduce } \Phi\ \} \\ & (a = \mathsf{assign}.(d.a.b).(\Phi.a.b.(r.a.b))) \qquad\qquad\qquad\qquad \wedge \\ & (\forall F :: (\forall m, b' : m \prec n \wedge b' \in N.b : \mathsf{ok}^m_{b'}.(F.b')) \Rightarrow \mathsf{ok}^n_b.(\Phi.a.b.F)) \end{array}$$

In the last step, we have assumed the existence of a function Φ, such that when applied to the array $r.a.b$, it yields an acceptable value for node b at round n. If such a function exists, then the problem is almost solved.

To summarize the calculation above, MD3.a can be refined to MD4 defined below. For all $a, b \in V$ and $n \in A$:

$$\text{MD4.a} : \quad {}_{\mathsf{MinDist}.a.b}\vdash\ \circlearrowright \mathsf{preOk}^n \tag{5.5.21}$$

$$\text{MD4.b} : \quad {}_{\mathsf{MinDist}.a.b}\vdash\ \circlearrowright (\mathsf{preOk}^n \wedge \mathsf{Ok}^n_b) \tag{5.5.22}$$

MD4.c : There exists a function Φ satisfying:

$$(\forall F :: \ (\forall m, b' : m \prec n \wedge b' \in N.b : \ \mathsf{ok}^m_{b'}.(F.b')) \Rightarrow \mathsf{ok}^n_b.(\Phi.a.b.F) \tag{5.5.23}$$

$$\text{MD4.d} : \quad \mathsf{assign}.(d.a.b).(\Phi.a.b.(r.a.b)) \ \in \ \mathsf{a}(\mathsf{MinDist}.a.b) \tag{5.5.24}$$

Let us now turn our attention back to MD3. The most important part of MD3 is MD3.a and MD3.b. The first has been refined to MD4 above. *Up to this point the results do not depend on the specific model of channels being used.* MD3.b requires convergence to cOk^n_b, which is the communication obligation of $\mathsf{MinDist}.a.b$ for round n. By (5.5.20) this means, at the very least, making the value of $r.a.b.c$ of all neighbors c to be acceptable for round n. What other obligation that remains depends on how the channels are being modelled. For example if we are using history variables to record messages that travel through the channels, we may have to require that eventually, not only the message that has just arrived, but also the messages which are still under way are all acceptable for the current round (and hence excluding the possibility that the convergence process is hindered by the presence of 'bad' messages in the channels). One may also want to expose some lower level communication protocol, such as the use of a 4-phase handshake protocol. The same remark also applies: one has to extend cOk to accommodate the obligation to stabilize every part of this protocol.

In this presentation we will take the simplest model for channels, namely the link-register model introduced in Section 5.4. The copy variable $r.a.b.c$ can be treated as a link-register between process $\mathsf{MinDist}.a.c$ and $\mathsf{MinDist}.a.b$. As the registers are the only means of communication, it will suffice to define cOk as in (5.5.20) but in which the $\Rightarrow$ is replaced by equality:

$$\mathsf{cOk}^n_b \ = \ (\forall c : b \in N.c : \mathsf{ok}^n_b.(r.a.c.b)) \tag{5.5.25}$$

Using the above definition, and a calculation which is much similar to the calculation for MD3.a, we can refine MD3.b to MD5 below. For all $a, b, c \in V$ such that $b \in E.c$, and $n \in A$:

$\mathsf{FSA} = ([\![a : a \in V : \mathsf{FSA}.a)$ where $\mathsf{FSA}.a = ([\![b : b \in V : \mathsf{FSA}.a.b)$ where $\mathsf{FSA}.a.b$ is defined as:

```
prog    FSA.$a.b$
read    $\{r.a.b.b' \mid b' \in V\} \cup \{r.a.c.b \mid c \in V\} \cup \{d.a.b\}$
write   $\{r.a.c.b \mid c \in V\} \cup \{d.a.b\}$
init    true
assign  $d.a.b := \Phi.a.b.(r.a.b)$
$[\![$       $([\![ c : c \in V \wedge b \in N.c : r.a.c.b := d.a.b)$
```

Figure 5.9: *Lentfert's FSA algorithm.*

$$\mathsf{MD5.a}: \quad {}_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n) \tag{5.5.26}$$

$$\mathsf{MD5.b}: \quad {}_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \wedge \mathsf{Ok}_b^n.(r.a.c.b)) \tag{5.5.27}$$

$$\mathsf{MD5.c}: \quad \mathsf{assign}.(r.a.c.b).(d.a.b) \in \mathsf{a}(\mathsf{MinDist}.a.b) \tag{5.5.28}$$

MD4.d and MD5.c already suggest what are the actions of MinDist.$a.b$. It remains to be verified that these actions respect all the stability requirements in MD4 and MD5, and that the MinDist.$a.b$'s are pair-wise write-disjoint. This is still a lot of work, but does not require insight, so we will leave this part out. Without detailing the verification, in Figure 5.9 is we present algorithm which will satisfies MD4, MD5, and other remaining specifications, provided MD4.c is satisfied. The name MinDist is replaced by FSA though, and indeed, it is the FSA algorithm mentioned so often before.

Note that the two versions of MinDist given early in this section have the same structure as the FSA algorithm in Figure 5.9. In fact, the latter is more general. The reader may have noticed that nowhere in our calculation —incomplete though it was— did we refer to any specific property of Dist. In fact, the algorithm in Figure 5.9 can be used to compute any function Δ of a compatible type, as long as one can find a predicate ok, a well-founded relation $\prec$, and a function Φ that satisfy MD4.c, and ok is such that (5.5.13) is satisfied, with Dist substituted by Δ of course. The condition stated in MD4.c is called *round solvability* condition, due to Lentfert in [Len93]. A function $\Delta \in V \rightarrow V \rightarrow B$ is called round solvable if one can find some ok, $\prec$, and Φ satisfying MD4.c. So, as a general result, the FSA algorithm self-stabilizingly computes any round solvable function[7]. *Whether or not a function is round solvable is, one can say, a purely mathematical property of the function.* The structure of the algorithm itself does not depend on it.

[7] The idea behind the name 'round solvable' is that a round solvable function can be self-stabilizingly computed by 'traversing' the rounds. The function Φ abstracts the computation required before advancing to the next round.

The name FSA (Fair and Successive Approximation) algorithm is inspired by the successive applications of Φ, which the algorithm basically does, and which can be thought as an attempt to approximate a fix point of Φ —that is, if Φ has one. In addition, the Φ belonging to $\mathsf{FSA}.a.b$ is typically parameterized by a and b. So, we actually have a family of functions. The fact that FSA is a UNITY program means that the order in which the Φ's are applied does not matter, as long as the order is fair.

We can also view the FSA algorithm in a slightly different way. Forget about the Δ. If we trace the calculation, the reader may notice that MinDist satisfies $\mathsf{true} \rightsquigarrow \mathsf{dataOk}^{\eta}$, for some η. Whatever the objective of the algorithm, it suffices to show that it is implied by dataOk^{η}. This is indeed a slightly more general view. For example, it is useful if instead of converging to $d = \Delta$, the program is required to converge to $d \in \{\Delta_i \mid i \in U\}$.

Figure 5.10 displays an overview of the relation between the initial specification of MinDist, MD0, and the various other MD's obtained from the previous calculation. Except for those concerning MD5, the refinement relations displayed in the figure do not depend on the choice of how channels are being modelled. As the algorithm satisfies MD0, it is therefore self-stabilizing: regardless of the initial values of r and d, it will converge to $d = \Delta$. It can also be made insensitive to changes in the network topology N by making N a program variable instead of a constant. Or course only if N defines a finite and connected network then the algorithm is guaranteed to converge to $d = \Delta$.

5.5.1 The Choice of ok and Φ

We have stated that Lentfert's FSA algorithm in Figure 5.9 is general enough to compute any round solvable function Δ. But let us turn our attention back to the minimal distance problem. A question that is still to be answered is whether the algorithm can self-stabilizingly compute the minimal distance function Dist. To answer this question we need to come up with concrete ok, $\prec$, and Φ, which so far are only partially specified by (5.5.13) and MD4.c.

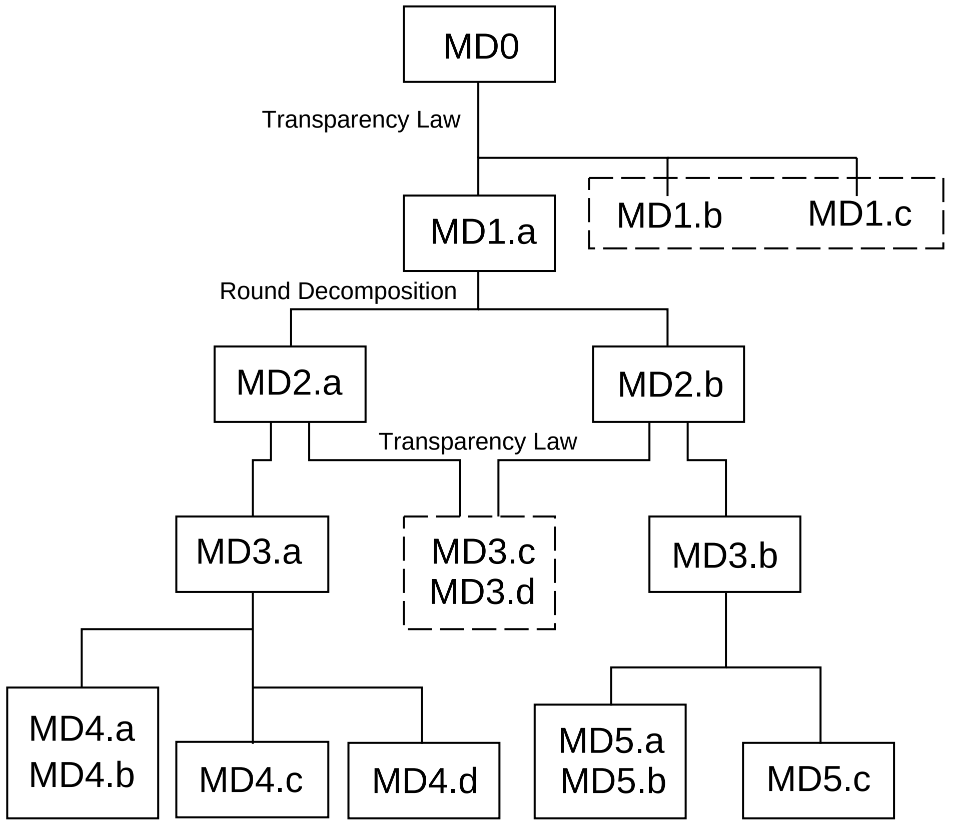
The definition of Dist is given by (5.5.2) and (5.5.3). The function defines a simple notion of minimum distance: the minimum distance between a and b is simply the minimum length of the paths between a and b. However, we are also interested in a more general notion of minimum distance as given by $(2.3.2)_{18}$. In this subsection a choice for ok, $\prec$, and Φ will be motivated. We will first take a look at the case of Dist, and then generalize the result.

Inspired by equations (5.5.2) and (5.5.3) the reader may have guessed the Φ required by the algorithm in Figure 5.9 in order to compute Dist. This is the function:

$$\mathsf{DistGen}.a.b.F \;=\; \begin{cases} 0 & \text{, if } a = b \\ \mathsf{min}\{F.b' + 1 \mid b' \in N.b\} & \text{, if } a \neq b \end{cases} \tag{5.5.29}$$

where F is a function of the type, here, $V \rightarrow \mathbb{N}$. In the algorithm in Figure 5.9 this is to be replaced by $r.a.b$.

We can generalize the above so that the FSA algorithm computes a general minimal distance function δ by replacing 0, the operator min, and the function $+1$ in the above



The dash-boxed specifications concern the write-disjunction of the components of FSA. The shaded-boxed specifications concern the stability of some predicates.

Figure 5.10: *A refinement scheme of the self-stabilizing minimal distance algorithm.*

with, respectively, the bottom element $\perp$ of some lattice $\sqsubseteq$, the greatest lower bound operator $\sqcap$ of $\sqsubseteq$, and a function addW:

$$\varphi.a.b.F \;=\; \begin{cases} \perp & \text{, if } a=b \\ \sqcap\{\mathsf{addW}.b'.b.(F.b') \mid b' \in N.b\} & \text{, if } a \neq b \end{cases} \tag{5.5.30}$$

For example, one may let the cost of the links in the network vary (instead of being always 1). Let say that the weight of the link from a to b is given by $w.a.b$. The definition of φ becomes:

$$\varphi.a.b.F \;=\; \begin{cases} 0 & \text{, if } a=b \\ \mathsf{min}\{F.b' + w.b'.b \mid b' \in N.b\} & \text{, if } a \neq b \end{cases} \tag{5.5.31}$$

If the identity of a neighbor —so-called best neighbor— through which a path with the minimal distance runs has also to be recorded, we can extend the result of φ[8] to have the type $\mathbb{N} \times V$. So, when applied to a function F, the function yields a pair (x, c)

[8] The function φ has a general type of $(V \to A) \to A$.

where x is the newly computed distance and c is the identity of a best neighbor. The definition of φ:

$$\varphi.a.b.F \;=\; \begin{cases} 0, \perp_V & \text{, if } a = b \\ \sqcap\{(F'.b' + w.b'.b, b') \mid b' \in N.b\} & \text{, if } a \neq b \end{cases} \tag{5.5.32}$$

Here, it is also crucial that the relation on which $\sqcap$ is based on is a lexicographic product of a lattice over $\mathbb{N}$ and a lattice over V[9]. Best neighbor computation will be exposed more thoroughly in Chapter 6.

The problem is however not only in finding a right Φ, but also in finding a right ok. One of the requirements —in fact, the first requirement we came up with— for ok is (5.5.13), stating that dataOk^η should imply $(\forall b : b \in V : d.a.b = \mathsf{Dist}.a.b)$, given that $\eta \in A$. Let us now do some calculation to see what ok should be:

$$\begin{array}{ll}
 & \mathsf{dataOk}^\eta \;\Rightarrow\; (\forall b : b \in V : d.a.b = \mathsf{Dist}.a.b) \\
= & \quad \{ \text{ definition}_{102} \text{ of } \mathsf{dataOk} \ \} \\
 & (\forall b : b \in V : \mathsf{Ok}_b^\eta) \;\Rightarrow\; (\forall b : b \in V : d.a.b = \mathsf{Dist}.a.b) \\
\Leftarrow & \quad \{ \text{ predicate calculus } \} \\
 & (\forall b : b \in V : \mathsf{Ok}_b^\eta \Rightarrow (d.a.b = \mathsf{Dist}.a.b)) \\
\Leftarrow & \quad \{ \ (\dagger) \text{ predicate calculus } \} \\
 & (\forall b, n : b \in V \wedge n \in A : \mathsf{Ok}_b^n = ((\mathsf{Dist}.a.b \preceq n) \Rightarrow (d.a.b = \mathsf{Dist}.a.b))) \ \wedge \\
 & (\forall a, b : a, b \in V : \mathsf{Dist}.a.b \preceq \eta) \\
\Leftarrow & \quad \{ \text{ definition}_{102} \text{ of } \mathsf{Ok} \ \} \\
 & (\forall b, n, X :: \mathsf{ok}_b^n.X = ((\mathsf{Dist}.a.b \preceq n) \Rightarrow (X = \mathsf{Dist}.a.b))) \ \wedge \\
 & (\forall a, b : a, b \in V : \mathsf{Dist}.a.b \preceq \eta)
\end{array}$$

Recall that in (5.5.7), the only specification that we have for ok is that $\mathsf{ok}_b^n.X$ means that X an 'acceptable' value for node b at round n. The calculation above suggests when an X is considered 'acceptable', namely if it is equal to the actual minimal distance between a and b, at least, it is so if $\mathsf{Dist}.a.b \preceq n$. More precisely:

$$\mathsf{ok}_b^n.X \;=\; (\mathsf{Dist}.a.b \preceq n) \Rightarrow (X = \mathsf{Dist}.a.b) \tag{5.5.33}$$

and hence Ok becomes:

$$\mathsf{Ok}_b^n.(d.a.b) \;=\; (\mathsf{Dist}.a.b \preceq n) \Rightarrow (d.a.b = \mathsf{Dist}.a.b)$$

This is not quite surprising. Earlier we have remarked that our strategy is to 'stabilize' the $d.a.b$'s round by round. So, when the final round η is reached, all $d.a.b$'s contain the correct value. The above is just the same strategy expressed in formulas.

Notice that a choice has been made in the step marked with $(\dagger)$ in the calculation above, namely that the domain of rounds A is required to include the range of $\mathsf{Dist}.a$.

[9] In addition, as we will see latter in Chapter 6, the operator $+$ is required to have $\top$ as its unit element.

Note also that η is required to be larger than any $\mathsf{Dist}.a.b$. But this is well expected as η abstracts the diameter of the network.

The above definition of ok_b^n is *almost* good. Let us see if we can fulfil MD4.c. For the sake of simplicity, we will take the simple notion of minimal distance as characterized by equations (5.5.2) and (5.5.2). As the domain of rounds, $[0 \dots n_{\mathsf{max}}]$ is taken, ordered by $<$. The definition of Φ is as in (5.5.29). Let now, $a, b \in V$ such that $\mathsf{Dist}.a.b = n+1$. From equation (5.5.3) one can conclude that $n \leq \mathsf{Dist}.a.b'$ holds for any neighbor b' of b, and some neighbors will have $\mathsf{Dist}.a.b' = n$. Let $Z = \{b' \mid (\mathsf{Dist}.a.b' = n) \wedge b' \in N.b\}$. MD4.c requires:

$$\begin{array}{ll} & \mathsf{ok}_b^{n+1}.(\mathsf{DistGen}.F) \\ = & \quad \{ \text{ definition of } \mathsf{DistGen} \text{ and } \mathsf{ok} \ \} \\ & \mathsf{min}\{F.b' + 1 \mid b' \in N.b\} \ = \ \mathsf{Dist}.a.b \end{array}$$

given that for all $m < n$ and $c \in V$, $\mathsf{ok}_c^n.(F.c)$ holds. This means that for the neighbors $b' \in Z$, $F.b'$ has a 'correct' value. That is, $F.b' = \mathsf{Dist}.a.b'$. However, nothing can be concluded about the neighbors $b'' \in N.b \backslash Z$. If one of those b'' has $F.b'' < n$ then the above equation will not hold. However, if:

$$(\mathsf{Dist}.a.b' \leq n \ \Rightarrow \ (F.b' = \mathsf{Dist}.a.b')) \ \wedge \ (n \leq \mathsf{Dist}.a.b' \ \Rightarrow \ n \leq F.b') \qquad (5.5.34)$$

for all neighbors b', this will suffice for $\mathsf{min}\{F.b' + 1 \mid b' \in N.b\}$ to yield $\mathsf{Dist}.a.b$.

The above suggests the following, more general, definition of ok. For all $b \in V$ and $n \in A$:

$$\mathsf{ok}_b^n.X \ = \ \begin{cases} X = \delta.a.b & , \text{if } \delta.a.b \preceq n \\ n \preceq X & , \text{if } n \preceq \delta.a.b \end{cases} \qquad (5.5.35)$$

Notice that (5.5.34) and (5.5.29), detailing the generator and the predicate ok for the simple minimal distance function is an instance of (5.5.29) and (5.5.35) above, obtained by replacing $\preceq$, $\sqcap$, and addW with $\leq$, min, and $+1$. Notice that min happens to be the greatest lower bound operator of $\leq$. This suggests a choice for $\preceq$, namely the (complete) lattice to which $\sqcap$ belongs. Of course, by MD4.c the non-reflexive part of $\preceq$, namely $\prec$, is still required to be well-founded. Recall that A, the domain of $\prec$ has been required to be finite. This is crucial, because otherwise we cannot apply the ROUND DECOMPOSITION$_{93}$ law used to break MD1.a to MD2. If $\preceq$ is a lattice over A, and it is $\sqcap$-closed, or if A is finite, then $\prec$ is well-founded. Recall also that in decomposing MD1.a to MD2 it is required that the minimal distance between any pair of nodes is bounded by some η —the diameter of the network. It has been remarked before that with the choice of generator and ok as in (5.5.29) and (5.5.35) the domain of rounds A is necessarily the same as the domain of distances. If A is a lattice then $\top$, the greatest element, exists and therefore can be selected for η.

Under some condition on $\sqcap$ and φ, ok and φ as defined by (5.5.35) and (5.5.30) satisfy MD4.c. The condition and its proof will however be postponed until Chapter 6.

Theorem 5.5.1 LENTFERT'S GENERAL THEOREM *FSA_sat_MDC*

Let (V, N) be a network with non-empty and finite V. Let A be a non-empty and finite domain (of rounds). If there exist a well-founded and transitive relation $\prec$ over A, a predicate $\mathsf{ok} \in A \to V \to V \to B \to \mathbb{B}$, and a function $\Phi \in V \to V \to V \to (V \to B) \to B$, such that they satisfy:

$$(\forall m, b' : m \prec n \land b' \in N.b : \mathsf{ok}^m_{a,b'}.(F.b')) \Rightarrow \mathsf{ok}^n_{a,b}.(\Phi.a.b.F)$$

for all $n \in A$, $a, b \in V$, and $F \in V \to B$, then the program FSA in Figure 5.9 satisfies:

$$\mathsf{true} \vdash \mathsf{true} \rightsquigarrow (\forall n, a, b : n \in A \land a, b \in V : \mathsf{ok}^n_{a,b}.(d.a.b))$$

Corollary 5.5.2

Let (V, N) be a network with non-empty and finite V. Let A be a non-empty and finite domain of rounds. Let $\mathsf{ok} \in A \to V \to V \to A \to \mathbb{B}$ and $\Phi \in V \to V \to V \to (V \to A) \to A$ be defined as:

$$\begin{aligned} \mathsf{ok}^n_{a,b}.X &= (\Delta.a.b \sqsubseteq n \Rightarrow (X = \Delta.a.b)) \land (n \sqsubseteq \Delta.a.b \Rightarrow n \sqsubseteq X) \\ \varphi.a.b.F &= \begin{cases} \bot & \text{, if } a = b \\ \sqcap\{\mathsf{addW}.b'.b.(F.b') \mid b' \in N.b\} & \text{, if } a \neq b \end{cases} \end{aligned}$$

where $\Delta \in V \to V \to A$, $\sqsubseteq$ is a lattice over non-empty and finite domain A, and $\sqcap$ is the greatest lower bound operator that belongs to $\sqsubseteq$. If $\sqsubset$, ok, and φ satisfy the condition stated in Theorem 5.5.1, then the program FSA in Figure 5.9 satisfies:

$$\mathsf{true} \vdash \mathsf{true} \rightsquigarrow (\forall a, b : a, b \in V : d.a.b = \Delta.a.b)$$

◀

Figure 5.11: *The convergence properties of the FSA algorithm.*

5.5.2 Conclusion

As a conclusion, in Figure 5.11 we present two theorems that summarize the results so far regarding the FSA algorithm. The first theorem is more general. It states where the algorithm can be expected to converge, and a condition —the *round solvability condition*— required to guarantee the convergence. The second theorem is specialized for a self-stabilizing computation of minimum-distance-like functions. The typing of some objects will be extended to accommodate some parameters which so far were kept implicit. We will also call a function δ satisfying the condition stated in the second theorem, that is, Corollary 5.5.2, as a *round solvable* function.

As a final note, the reader may notice that nowhere in the previous calculation a reference is made to the initial assumption that the network (V, N) is bi-directional. Indeed, this is not required. However, suppose we want to compute Dist. The value of Dist.$a.b$ is to be stored in the variable $d.a.b$, which is maintained by the program FSA.$a.b$, which is to be allocated in node b. To do its computation, this program requires information from all neighbors of b, so links must exist *from* all nodes in $N.b$ *to* b. The intention of computing minimal distance is so that we can send messages along some path yielding the minimal distance. That is, from b to its best neighbor, and so on, until the destination is reached. This means that there should be a link *to* the best neighbor of b *from* b. In other words, we need bi-directional links.

You alone,
Can make my song take flight.
It's over now,
The music of the night ...
—the Phantom

Chapter 6

Round Solvability of Cost Functions

The most important condition for the FSA algorithm, when applied to the minimum distance problem, is that the minimum distance function satisfies the round solvability condition. A general notion of distance will be introduced, and its round solvability will be investigated. The generalization is useful to handle a more sophisticated notion of distance.

IN Chapter 5, the FSA algorithm —displayed in Figure 5.9— was derived. It was stated in Theorem 5.5.1 that the algorithm can self-stabilizingly compute any round solvable function. To prove the round solvability of any function, it is required that we come up with a well-founded relation $\prec$, a predicate ok, and a function Φ satisfying the (round solvability) condition stated in Theorem 5.5.1. The round solvability of a function is unfortunately not a trivial condition. For minimal-distance-like functions, in which we are particularly interested, a choice for these $\prec$, ok, and Φ was proposed in Corollary 5.5.2. The choice has been motivated in Subsection 5.5.1. It remains now to prove that this choice is indeed a good one.

Instead of focusing on some particular minimum distance functions, such as the simple minimum distance function defined in $5.5.2_{101}$ and $5.5.3_{101}$, we are going to consider *minimum-distance-like* functions. How one defines the 'minimal distance' between two nodes depends on several things:

i. how one defines the cost of a link.

ii. how one 'sums' the link-costs to obtain the cost of a path.

iii. how one defines the 'minimum' of path-costs to obtain the minimum cost or distance.

In the simple minimum distance function, the link-costs are simply the constant 1. A more flexible function can be obtained by allowing the link-costs to take different values instead of simply 1. In practice, one may want to define the cost of a link as a vector, say, (c, s, r, i) where c tells us about the cost of using the link in, say, \$/kbit data sent, s tells us the capacity of the link in kbit/sec, r is some measure of the reliability of the link, and i is some information describing the identity or the status of the source node of the link. If the link-costs are all 1 —or at least, they are natural numbers—, we will have a good idea of how to 'sum' them up. For vectors, we usually take a point-wise summation. Summing the c-components in the above example probably means summing them using the usual '+' function. Summing the s-components may mean taking the minimum or the average of the values involved. How to sum up reliability measures r and the source node descriptions i is not quite clear and will depend on the way we are using the information. Because of this diversity we feel that it is better to provide a general theory for the round solvability of minimal-distance-like functions[1].

We have given equations (5.5.2) and (5.5.3) as the definition for the minimal distance function Dist. The equations do not really tell whether Dist.$a.b$ will indeed return the minimum length of the paths between a and b, that is, what is considered as the intuitive or operational meaning of Dist. In this case the relation between the operational meaning and the more abstract definition is quite clear. However, when a general notion of distance is being considered, one may ask in what respect it reflects our intuitive idea about distances. Instead of generalizing the equations (5.5.2) and (5.5.3) we will start by generalizing the operational meaning of (minimum) distance. Then, it will be investigated under what conditions it satisfies the properties that we expect a decent notion of distance to satisfy, and under what conditions it is round solvable. Although the general properties of a minimum-distance-like function are probably not surprising, it is noteworthy that the condition required for the round solvability of such a function is not much harder than the condition required to satisfy some of the most basic properties of minimum distance.

If the readers recall, to compute the simple minimal distance function Dist using the FSA algorithm we instantiate the function Φ in the algorithm with DistGen defined in $(5.5.29)_{107}$. This function DistGen happens to satisfy:

$$\mathsf{Dist}.a.b \ = \mathsf{DistGen}.a.b.\mathsf{Dist}$$

The function DistGen is also called a *generator*[2]. The above states that Dist is a fix point of its generator —or more precisely of: $(\lambda f, a, b.\ \mathsf{DistGen}.a.b.f)$. One may ask whether this fix point always exists for any round solvable function, and whether the round solvability condition actually defines a unique fix point. This will also be investigated in this chapter. Also, round solvability turns out to be a quite strong condition. Given a generator Φ satisfying the (round solvability) condition in Theorem 5.5.1, one can show that it generates, under some reasonable condition, a unique function. Or, in other words, the round solvability of a function characterizes the function.

In addition, two special extensions —or instances, depending on how one looks at them— of minimal distance functions will be discussed. The first extension can be used to record the best neighbor, that is, a neighbor through which a best path goes. The second extension can be used to broadcast data. Both extensions yield minimum-distance-like functions again, and hence the general theory about such functions can be applied.

[1] As the name implies, the round solvability of a minimal-distance-like function can be argued —if we are being very lazy— in a 'similar' way as the simple minimal distance function. However, let us remind the reader, in mechanical verifications argument such as *'can be proven in a more or less similar way'* is not adequate, if at all recommended.

[2] Because it is used by the FSA algorithm to generate Dist.

6.1 Basic Lattice Theory

In this section we give an overview of notions and facts —mainly about lattices— which are useful later. The reader may wish to skip this section if he feels familiar with lattice theory, and later consult it when necessary.

Let $\sqsubseteq$ be a relation on a type A. $\sqsubseteq$ is called a *partial order* if it is reflexive, transitive, and anti-symmetric. An m is said to be a *lower bound* of B, a subset of A, if $(\forall n : n \in B : m \sqsubseteq n)$ holds. The greatest lower bound of B, denoted by $\sqcap B$, does not necessarily exist for any B. If $\sqsubseteq$ is a partial order (over A) and $\sqcap B$ does exist for any subset B of A, then $\sqsubseteq$ is said to form a *lattice*[3]. The $\sqcap$ of A and $\emptyset$ are usually denoted by $\bot$ and $\top$. If they exist, they correspond with, respectively, the least and greatest element of A.

$\sqcap\{a, b\}$ is also denoted by $a \sqcap b$. If $\sqsubseteq$ is a lattice, this binary $\sqcap$ is idempotent, commutative, associative, and has $\top$ as its unit element. Actually, if such a binary operator is given, it defines a lattice by defining $m \sqsubseteq n = (m = m \sqcap n)$. For the set-level $\sqcap$, its implicit commutativity and associativity are expressed by (5.3.8).

The non-reflexive part of a $\sqsubseteq$ is denoted by $\sqsubset$. It is equal to $\sqsubset - I$[4]. A lattice $\sqsubseteq$ is said to be $\sqcap$-*closed* if $\sqcap B \in B$, for all non-empty B. A $\sqcap$-*closed* lattice is *linear*, meaning that for any pair (x, y), we have either $x \sqsubseteq y$ or $y \sqsubseteq x$. Usually, the 'minimum' of a minimum distance function corresponds to the $\sqcap$ operator of a $\sqcap$-closed lattice. Due to its linearity, such a lattice is perhaps not a very exciting lattice to discuss. However, as said, a lattice is not the only component we need to define a minimum distance function, and to investigate the exact role of the other components some amount of lattice theory will be helpful.

If $\sqsubseteq$ is a $\sqcap$-closed lattice, then $\sqsubset$ is *well-founded*. $\sqsubseteq$ and $\sqsubset$ will satisfy:

$$(m \not\sqsubseteq n = n \sqsubset m) \;\wedge\; (m \not\sqsubset n = n \sqsubseteq m) \qquad (6.1.1)$$

$$(\forall m' : m' \sqsubset m : m' \sqsubset n) \;=\; m \sqsubseteq n \qquad (6.1.2)$$

A function $f \in A \to A$ is said to be $\sqcap$-*distributive* if $f(\sqcap B) = \sqcap(f * B)$ for all $B \subseteq A$. If it is only distributive over all *non-empty* subsets of A, then f is said to be $\sqcap$-junctive[5]. A $\sqcap$-distributive function is also monotonic. If f is $\sqcap$-junctive and $f.\top = \top$ then f is $\sqcap$-distributive.

The *lexicographic* product of two relations $\sqsubseteq_1$ and $\sqsubseteq_2$, denoted by $\sqsubseteq_1 \bigotimes \sqsubseteq_2$, is

[3] Usually, such a partial order is called a *complete* lattice. The term 'lattice' is reserved for a partial order in which only the $\sqcap$ of any pair $\{x, y\}$ needs to exists. However, in this thesis all lattices considered are complete, and we do not feel it necessary to mention the attribute 'complete' all the time.

[4] I denotes the identity relation: $I.x.y = (x = y)$.

[5] Another commonly used and closely related notion is continuity. Continuity requires f to be distributive only over subsets of A that form 'chains'. It must be said that the literature often disagrees on the exact definition of distributivity, junctivity, and continuity.

Definition 6.1.1 Partial Order *PO*

$$\sqsubseteq \text{ is a partial order } = \begin{cases} (\forall n :: n \sqsubseteq n) & \wedge \\ \mathsf{Trans}.\sqsubseteq & \wedge \\ (\forall n, m :: n \sqsubseteq m \wedge m \sqsubseteq n \Rightarrow (m = n)) \end{cases}$$

Definition 6.1.2 Lower Bound *is_LB*

$$m \text{ is a lower bound of } B \;=\; (\forall n : n \in B : m \sqsubseteq n)$$

Definition 6.1.3 Greatest Lower Bound *isCAP*

$$(n = \sqcap B) \;=\; \begin{cases} n \text{ is a lower bound of } B & \wedge \\ (\forall m : m \text{ is a lower bound of } B : m \sqsubseteq n) \end{cases}$$

Definition 6.1.4 Lattice *CAP_PLa*

$$\mathsf{Lattice}.\sqsubseteq \;=\; \sqsubseteq \text{ is a partial order } \wedge (\forall B :: (\exists n :: n = \sqcap B))$$

Definition 6.1.5 $\sqcap$-Closed *CAP_Closed*

$$\sqsubseteq \text{ is } \sqcap\text{-closed} \;=\; (\forall B :: B \neq \emptyset : \sqcap B \in B)$$

Definition 6.1.6 $\sqcap$-distributivity *CAP_Distr*

$$f \text{ is } \sqcap\text{-distributive} \;=\; (\forall B :: f(\sqcap B) = \sqcap(f * B))$$

Definition 6.1.7 $\sqcap$-junctivity *CAP_Junct*

$$f \text{ is } \sqcap\text{-junctive} \;=\; (\forall B : B \neq \emptyset : f(\sqcap B) = \sqcap(f * B))$$

◀

Figure 6.1: *Some definitions from lattice theory.*

defined by:

$$\begin{array}{c} (m_1, m_2)\ (\sqsubseteq_1 \bigotimes \sqsubseteq_2)\ (n_1, n_2) \\ = \\ ((m_1 \neq n_1) \wedge m_1 \sqsubseteq_1 n_1) \ \vee\ ((m_1 = n_1) \wedge m_2 \sqsubseteq_2 n_2) \end{array} \tag{6.1.3}$$

If both $\sqsubseteq_1$ and $\sqsubseteq_2$ are lattices, then so is their lexicographic product. The construction also preserves the $\sqcap$-closedness of the lattices involved. If we denote the $\sqcap$ which corresponds to $\sqsubseteq_1$ with $\sqcap_1$, the one to $\sqsubseteq_2$ with $\sqcap_2$, and the one to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$ with $\sqcap_3$, it can be shown that $\sqcap_3$ is characterized by:

$$\sqcap_3 A \;=\; (\sqcap_1(\mathsf{fst} * A), \sqcap_2\{\mathsf{snd}.n \mid n \in A \wedge (\mathsf{fst}.n = \sqcap_1(\mathsf{fst} * A))\}) \tag{6.1.4}$$

A summary of the above definitions is given in Figure 6.1.

6.2 Paths and Cost of Paths

In this section the notion of a path through a network will be formally defined, and subsequently general notions of distance and minimum distance over paths will also be defined and their basic properties discussed. To emphasize the distinction with the simple notion of distance, in the sequel we will use the term *cost* instead of distance.

As said, a network (graph) is a pair (V, N) where V is the set of nodes in the network, and N is a function describing the connectivity of the network. That is, $N.a$ is the set of all nodes in V to which a is connected to. We call the members of $N.a$ *neighbors*[6] of a. Naturally, we require $(\forall a : a \in V : N.a \subseteq V)$.

We can define the reflexive and transitive closure of the *neighborhood function* N in the usual way: $N^*.a = \cup\{N^n.a \mid n \in \mathbb{N}\}$ where $N^0.a = \{a\}$ and $N^{n+1} = N \circ N^n$. A node b is said to be *reachable* from a if $b \in N^*.a$. A network is said to be *connected* if all nodes in the network are pair-wise reachable.

Definition 6.2.1 CONNECTED NETWORK *Connected*

$$\mathsf{connected}(V, N) \;=\; (\forall a, b : a, b \in V : a \in N^*.b)$$

In the sequel, let (V, N) be a network.

There are several common ways to define a path in a network. One of them is as a non-empty sequence of nodes such that each node in the sequence is a neighbor of the previous node. The first node is the destination of the path, and the last node is the source (or the other way around, depending on how one orders the nodes). We will use a slightly different definition[7]:

Definition 6.2.2 PATH *PATH*

$$\begin{aligned} U \xrightarrow{[]} b \;&=\; b \in U \\ U \xrightarrow{b';s} b \;&=\; b' \in N.b \;\wedge\; U \xrightarrow{s} b' \end{aligned}$$

So, $U \xrightarrow{s} b$ means that $b;s$ is a path —in the sense of the above conventional definition— from U to b[8]. The source (U) is generalized by allowing it to be a set instead of a single node. A path has U as its source if it starts somewhere in U. One can show that a path defined as above satisfies its intended definition:

[6] This term can be misleading. One tends to think that neighborhood is a symmetric relation. In discussing the program MinDist in Chapter 5, the network is assumed to be bi-directional, in which case the neighborhood relation is symmetric indeed. In general however, this should not be assumed.

[7] The only reason is that at the moment we defined it in HOL, it seemed convenient.

[8] Or a path from b to U, depending on how one interprets the 'direction' induced by the neighborhood function N.

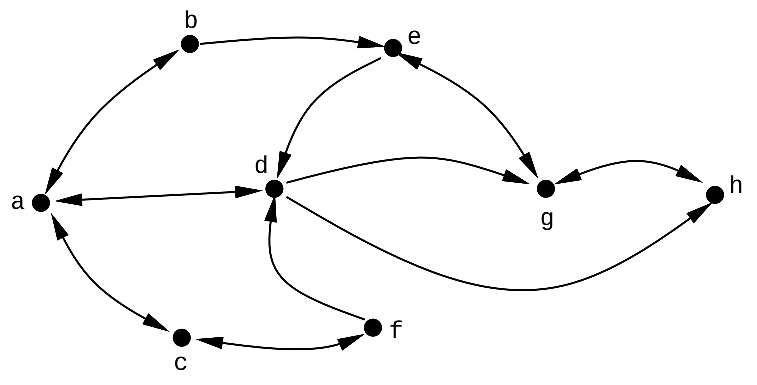


Figure 6.2: *A connected network*

$$U \xrightarrow{s} b \;\Rightarrow\; (\forall i : i < \ell.s : s.i \in N.((b;s).i)) \tag{6.2.1}$$
$$U \xrightarrow{s;a} b \;\Rightarrow\; a \in U \tag{6.2.2}$$

It also satisfies the following 'split' property:

$$U \xrightarrow{s} b \;=\; (\exists b',t :: (s = b';t) \wedge b' \in N.b \wedge U \xrightarrow{t} b') \;\vee\; (s = [] \wedge b \in U) \tag{6.2.3}$$

Figure 6.2 shows a connected but non-bi-directional network. An arrow from a node x to y means $x \in N.y$. An example of a path is $s = d;e;b;a$ which is a path from $\{a,c\}$ to g and to h. That is, $\{a,c\} \xrightarrow{s} g$ and $\{a,c\} \xrightarrow{s} h$.

The connectedness of a network can also be expressed in terms of paths:

$$\mathsf{connected}(V,N) \;=\; (\forall a,b : a,b \in V : (\exists s :: \{a\} \xrightarrow{s} b)) \tag{6.2.4}$$

The cost of a path, is simply the 'sum' of the cost of the links along the path:

Definition 6.2.3 PATH-COST *PathCost*

$$\begin{array}{rcl} \varrho.b.[] & = & e \\ \varrho.b.(b';s) & = & \mathsf{addW}.b'.b.(\varrho.b'.s) \end{array}$$

for some $e \in A$, $\mathsf{addW} \in V \to V \to A \to A$, and type A.

The function addW abstracts the summation of the cost of the links along the path. For the simple notion of cost we have $e = 0$ and $\mathsf{addW}.b'.b.X = 1 + X$, and hence $\varrho.b.s = \ell.s$.

The minimum cost of going from a source U to a destination b is simply the greatest lower bound, with respect to an ordering $\sqsubseteq$, of the cost of all paths from U to b:

Definition 6.2.4 MINIMUM COST *MiCost*

$$\delta.U.b \;=\; \sqcap\{\varrho.b.s \mid U \xrightarrow{s} b\}$$

where $\sqcap$ belongs to some relation $\sqsubseteq$ over A.

Since there can be infinitely many paths in a network (even if the network is finite) exhaustively computing δ is not likely to terminate. If the simple notion of cost (distance) is used, it is known that the corresponding minimal cost function can be computed in finite time by exploiting its recurrence relation (5.5.2) and (5.5.3). Let us see if a similar relation can be obtained for minimal cost functions in general.

Let in the calculation below $\sqsubseteq$ be a relation on which $\sqcap$ is based. To guarantee that $\delta.U.b = \sqcap\{\varrho.b.s \mid U \xrightarrow{s} b\}$ always exists, we insist that $\sqsubseteq$ is a lattice. Two cases can be distinguished, $b \in U$ and $b \notin U$.

If $b \in U$ and $e = \bot$, then $\delta.U.b = \bot$:

$$
\begin{array}{ll}
 & \delta.U.b = \bot \\
= & \quad \{ \text{ definition of } \delta \ \} \\
 & \sqcap\{\varrho.b.s \mid U \xrightarrow{s} b\} = \bot \\
\Leftarrow & \quad \{ \sqsubseteq \text{ is a lattice, } \bot \text{ is the least element } \} \\
 & (\exists s :: U \xrightarrow{s} b \wedge (\varrho.b.s = \bot)) \\
\Leftarrow & \quad \{ \text{ take } s = [] \ \} \\
 & U \xrightarrow{[]} b \wedge (\varrho.b.[] = \bot) \\
= & \quad \{ \ b \in U, \text{ definition of } \longrightarrow \text{ and } \varrho \ \} \\
 & e = \bot
\end{array}
$$

If the minimum cost to all neighbors of b is known, then the minimum cost to b can be computed by applying addW to the minimum cost to each neighbor, and combine the results using $\sqcap$. However, to be able to do this, several things are required. One of them is that addW is $\sqcap$-distributive. The following calculation will show it. Let b such that $b \notin U$.

$$
\begin{array}{ll}
 & \delta.U.b \\
= & \quad \{ \text{ definition of } \delta \ \} \\
 & \sqcap\{\varrho.b.s \mid U \xrightarrow{s} b\} \\
= & \quad \{ \ (6.2.3),\ b \notin U \ \} \\
 & \sqcap\{\varrho.b.(b';t) \mid b' \in N.b \wedge U \xrightarrow{t} b'\} \\
= & \quad \{ \text{ set theory } \} \\
 & \sqcap(\cup\{\{\varrho.b.(b';t) \mid U \xrightarrow{t} b'\} \mid b' \in N.b\}) \\
= & \quad \{ \ (5.3.8)_{97},\ \text{definition}_{95} \text{ of } * \ \} \\
 & \sqcap\{\sqcap\{\varrho.b.(b';t) \mid U \xrightarrow{t} b'\} \mid b' \in N.b\}
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ definition of } \varrho \text{ and } * \} \\
& \sqcap\{\sqcap(\mathsf{addW}.b'.b * \{\varrho.b'.t \mid U \xrightarrow{t} b'\}) \mid b' \in N.b\} \\
= & \{ \text{ } (\dagger) \text{ insist that } \mathsf{addW}.b'.b \text{ is } \sqcap\text{-distributive } \} \\
& \sqcap\{\mathsf{addW}.b'.b.(\sqcap\{\varrho.b'.t \mid U \xrightarrow{t} b'\}) \mid b' \in N.b\} \\
= & \{ \text{ definition of } \delta \text{ } \} \\
& \sqcap\{\mathsf{addW}.b'.b.(\delta.U.b') \mid b' \in N.b\}
\end{array}
$$

Notice that $\sqcap$-distributivity of addW is required in ($\dagger$). Alternatively, we can also require that addW is $\sqcap$-junctive, but then it must be guaranteed that the set $\{\varrho.b'.t \mid U \xrightarrow{t} b'\}$ is non-empty for all $b' \in N.b$. This is met, for example if the network is connected. There is however a good reason to require addW to be $\sqcap$-distributive. The function $\mathsf{addW}.b'.b$ is intended to add the cost of the link (b', b) to the so far computed path-cost. In the literature it is always required that link-cost is a non-zero positive number —this is crucial for the termination of any minimum cost algorithm. This assumption can be expressed in terms of addW as follows:

$$x \sqsubset \mathsf{addW}.b'.b.x \tag{6.2.5}$$

However, what will happen if x already hits the $\top$? Well, since we cannot go beyond $\top$ we do not have much choice but to impose that $\mathsf{addW}.b'.b.\top = \top$. This however means that instead of requiring addW to be $\sqcap$-junctive we can as well require it to be $\sqcap$-distributive.

The recurrence relation obtained from the above calculation is re-stated in Figure 6.3. In addition one can prove that for all neighbors b', adding the cost of the link (b', b) to $\delta.U.b$ should yield a value which is at least $\delta.U.b$. If the lattice $\sqsubseteq$ is also $\sqcap$-closed —and hence total— there also exists a neighbor $a \in N.b$ —so-called a *best neighbor*— such that $\delta.U.a$, plus the cost of the link (a, b), is exactly equal to $\delta.U.b$. Hence, there is a path with the optimal cost which goes from U to b, and passes a —which is why a is called a best neighbor. For example, using the simple notion of cost, the minimal cost from a to g in the network in Figure 6.2 is 2, and the best neighbor of g is d. The $\sqcap$-closedness is required because otherwise $\delta.U.b = \sqcap\{\mathsf{addW}.b'.b.(\delta.U.b') \mid b' \in N.b\}$ may not be an element of the set $\{\mathsf{addW}.b'.b.(\delta.U.b') \mid b' \in N.b\}$ and hence $\mathsf{addW}.a.b.(\delta.U.a)$ —where a is the best neighbor of b— may not be exactly $\delta.U.b$.

Theorems δ SELF and δ SPLIT in Figure 6.3, stating a recurrence relation for δ, do not indeed specifically tell us how to compute δ, but they do give an obvious suggestion. In Section 6.4 the round solvability of δ will be shown. Lentfert's FSA algorithm given in Figure 5.9 can then be used to self-stabilizingly compute this function[9].

6.2.1 Denoting the Context

Implicit in the definition of ϱ and δ are the relation $\sqsubseteq$ and the function addW which can be considered parameters of the definition. To carry them around all the time will

[9] Of course, we do not need round solvability just to show the computability of δ. Still, recall that self-stabilization is of a greater interest to this thesis.

Theorem 6.2.5 δ SELF *MiCost_SELF*

$$\frac{\textsf{Lattice}.\sqsubseteq\ \wedge\ (e=\bot)\ \wedge\ b\in U}{\delta.U.b=\bot}$$

Theorem 6.2.6 δ SPLIT *MiCost_SPLIT1*

$$\frac{\textsf{Lattice}.\sqsubseteq\ \wedge\ (\forall a,b::\textsf{addW}.a.b \text{ is } \sqcap\text{-distributive})\ \wedge\ b\notin U}{\delta.U.b\ =\ \sqcap\{\textsf{addW}.b'.b.(\delta.U.b')\mid b'\in N.b\}}$$

Theorem 6.2.7 δ LESS *MiCost_LESS_NEIGHBOR1*

$$\frac{\textsf{Lattice}.\sqsubseteq\ \wedge\ (\forall a,b::\textsf{addW}.a.b \text{ is } \sqcap\text{-distributive})\ \wedge\ b'\in N.b}{\delta.U.b\ \sqsubseteq\ \textsf{addW}.b'.b.(\delta.U.b')}$$

Theorem 6.2.8 BEST NEIGHBOR EXISTENCE *MiCost_IN_NEIGHBOR1*

$$\frac{\begin{array}{c}\textsf{Lattice}.\sqsubseteq\ \wedge\ (\forall a,b::\textsf{addW}.a.b \text{ is } \sqcap\text{-distributive})\\ \sqsubseteq \text{ is } \sqcap\text{-closed}\ \wedge\ b\notin U\ \wedge\ (N.b\neq\emptyset)\end{array}}{(\exists a:a\in N.b:\textsf{addW}.a.b.(\delta.U.a)=\delta.U.b)}$$

◄

Figure 6.3: *Some basic theorems of minimal cost functions.*

severely worsen the readability[10] of formulas. Still, in the sequel things will become more complicated as objects with a similar structure but based on different building blocks will have to be considered and compared. Therefore we introduce the following notation, which will only be used when necessary:

$$\alpha,\beta,\ldots\ \Vdash\ f$$

The objects to the left of $\Vdash$ are called the context of f. The meaning of f is to be interpreted by associating whatever implicit parameters it requires to the objects in the context. Care will be taken that no confusion will arise as to which parameter belongs to which object in the context. If it is clear from the surrounding text, it will not be necessary to bind all implicit parameters to the context. For example:

$$f,\sqsubseteq_1\bigotimes\sqsubseteq_2\Vdash\ \delta$$

is a δ function which is based on the function f (instead of addW) and the relation $\sqsubseteq_1\bigotimes\sqsubseteq_2$ (instead of $\sqsubseteq$).

6.3 Best Neighbor

In Theorem 6.2.8 it is stated that if the lattice $\sqsubseteq$ is $\sqcap$-closed, and if $b\notin U$, there always exists a best neighbor. That is, a neighbor through which the best path from U to b

[10] Albeit being fairly formal, we are still arguing at the human level where dropping parameters —or abusing notation in general— is a bad but a useful common practice. Once we are in HOL, all formulas have to be written completely.

passes. A frequently occurring task is to extend a minimal cost function that it also yields the best neighbor. The problem has been touched upon several times before. In this section we intend to give a more extensive analysis. To record the best neighbor, the type of the results of $\varrho.b.s$ is extended from A to $A \times V$. The second component will be used to record the node in s, just prior to the destination b (thus that node is $\mathsf{hd}.s$). That is, let us call the resulting function $\lceil\varrho\rceil$:

$$\begin{array}{lcl} \lceil\varrho\rceil.b.[] & = & (\varrho.b.[], e_2) \\ \lceil\varrho\rceil.b.(b';s) & = & (\varrho.b.(b';s), b') \end{array} \tag{6.3.1}$$

for some node $e_2 \in V$.

At the informal level this idea will sound quite natural. Expressed in formulas as above, things may be quite disorienting. We will however restrain from arguing the choice informally as this will prove to be lengthy. Instead, below we give a calculation that will expose the choice above. Still, the reader is encouraged to relate the above definition with his operational view towards the problem.

Let us define the corresponding minimal cost function as follows:

$$\lceil\delta\rceil.U.b \;=\; \sqcap_3\{\lceil\varrho\rceil.b.s \mid U \xrightarrow{s} b\} \tag{6.3.2}$$

where $\sqcap_3$ correspond to some lattice $\sqsubseteq_3$, the choice of which will be made apparent by the calculation below. Let now $b \notin U$. We derive:

$$\begin{array}{ll} & \lceil\delta\rceil.U.b \\ = & \quad \{\ (6.3.2)\ \} \\ & \sqcap_3\{\lceil\varrho\rceil.b.s \mid U \xrightarrow{s} b\} \\ = & \quad \{\ (\dagger)\text{ let } \lceil\varrho\rceil.b.s = (\varrho.b.s, f.s)\ \} \\ & \sqcap_3\{(\varrho.b.s, f.s) \mid U \xrightarrow{s} b\} \\ = & \quad \{\ (\ddagger)\text{ let } \sqsubseteq_3 = \sqsubseteq_1 \otimes \sqsubseteq_2,\ (6.1.3)\ \} \\ & (\sqcap_1\{\varrho.b.s \mid U \xrightarrow{s} b\},\ \sqcap_2\{f.s \mid U \xrightarrow{s} b \wedge (\varrho.b.s = \sqcap_1\{\varrho.c.s \mid U \xrightarrow{s} c\})\}) \\ = & \quad \{\ \text{definition}_{119}\text{ of } \delta\ \} \\ & (\delta.U.b,\ \sqcap_2\{f.s \mid U \xrightarrow{s} b \wedge (\varrho.b.s = \delta.U.b)\}) \end{array}$$

Note the choice for $\sqsubseteq_3$ was made in ($\ddagger$), namely $\sqsubseteq_3 = \sqsubseteq_1 \otimes \sqsubseteq_2$. It must also be assumed that $\sqsubseteq_1$ and $\sqsubseteq_2$ are lattices.

If we choose $f.s = \mathsf{hd}.s$, then the above derivation states that the second component of $\lceil\delta\rceil.U.b$ will be the first node of some path s to b, such that the cost of this path is minimal. The first node of s, by definition of path, is a neighbor of b. Hence we have obtained the best neighbor. Or at least, we hope so. A small complication arises if:

$$\sqcap_2\{\mathsf{hd}.s \mid U \xrightarrow{s} b \wedge \ldots\}$$

does not yield something which is an element of the set $\{\mathsf{hd}.s \mid U \xrightarrow{s} b \wedge \ldots\}$. This can be mended by requiring that $\sqcap B \in B$, if B is non-empty. That is, by requiring the lattice $\sqsubseteq_2$ to be $\sqcap$-closed.

In addition, one may also require that the minimal cost from U to b is *equal to* —instead of being the best approximation of— the minimal cost to the best neighbor b', combined with the cost of the link b' to b using $\mathsf{addW}.b'.b$. In this case, $\sqsubseteq_1$ also needs to be $\sqcap$-closed. This is a strong requirement, because a $\sqcap$-closed lattice has to be linear.

Let us now do some more calculation to rewrite $\lceil\varrho\rceil$:

$$
\begin{array}{ll}
 & \lceil\varrho\rceil.b.(b';s) \\
= & \quad \{\ (6.3.1)\ \} \\
 & (\varrho.b.(b';s), b') \\
= & \quad \{\ \text{definition}_{118} \text{ of } \varrho\ \} \\
 & (\mathsf{addW}.b'.b.(\varrho.b'.s),\ b') \\
= & \quad \{\ (\dagger) \text{ let } >\!\!\#.f.b'.b.(X,c) = (f.b'.b.X, b')\ \} \\
 & >\!\!\#.\mathsf{addW}.b'.b.(\varrho.b'.s,\ \mathsf{snd}.(\lceil\varrho\rceil.b'.s)) \\
= & \quad \{\ (6.3.1)\ \} \\
 & >\!\!\#.\mathsf{addW}.b'.b.(\lceil\varrho\rceil.b'.s)
\end{array}
$$

Now take a look at the definition$_{118}$ of ϱ: $\lceil\varrho\rceil$ satisfies the equations, only with a different choice for addW, namely $>\!\!\#.\mathsf{addW}$ (the *fish-tail* of addW). In other words, $\lceil\varrho\rceil$ is just an instance of ϱ, and consequently $\lceil\delta\rceil$ is also an instance of the δ function. Notice that this higher order function fish-tail is exactly the function required to extend a minimal cost function to a function that also records the best neighbors. The reader may want to compare the definition of $>\!\!\#.\mathsf{addW}$ as given in ($\dagger$) with an instance thereof in $(5.5.32)_{109}$.

This raises however a new question: if we extend the function addW and the lattice $\sqsubseteq_1$ on which a δ function is based to, respectively, $>\!\!\#.\mathsf{addW}$ and $\sqsubseteq_1 \bigotimes \sqsubseteq_2$, does the resulting δ function satisfies the general properties of a minimal cost function? If the reader goes over the properties in Figure 6.3 he will notice that those properties require a lattice to be used, the $\sqcap$-closedness of the lattice, and the distributivity of the function addW being used. The lexicographic product preserves lattices and the $\sqcap$-closedness. So, if $\sqsubseteq_1$ and $\sqsubseteq_2$ are both lattices, then so is $\sqsubseteq_1 \bigotimes \sqsubseteq_2$, and if both $\sqcap_1$ and $\sqcap_2$ are $\sqcap$-closed, then so is $\sqsubseteq_1 \bigotimes \sqsubseteq_2$. Unfortunately, the function $>\!\!\#.\mathsf{addW}$ as it is now is not $\sqcap$-distributive because $>\!\!\#.\mathsf{addW}.b'.b.(\top,\top) = (\mathsf{addW}.b'.b.\top, b')$, which is not necessarily equal to $\top$. To make $>\!\!\#.\mathsf{addW}$ distributive we will redefine it such that when applied to $\top$ it yields $\top$. This is displayed in Figure 6.4[11]. The distributivity of this new definition and some other facts mentioned above are also displayed as theorems in Figure 6.4.

[11] A naive solution is to define $>\!\!\#.\mathsf{addW}.b'.b.(X,c) = \top$ if $(X,c) = \top$, and otherwise $>\!\!\#.\mathsf{addW}.b'.b.(X,c) = (f.b'.b.X, b')$ as before. However this does not yield a function which satisfies the non-zero link-cost condition (6.2.5). That is, $((X,c) \neq \top) \Rightarrow (X,c) \sqsubset >\!\!\#.\mathsf{addW}.b'.b.(X,c)$ is not satisfied by this definition of $>\!\!\#$.

Definition 6.3.1 FISH-TAIL *FishTail*

For any $f \in V {\to} V {\to} (A \times V) {\to} (A \times V)$:

$$\rightarrowtail\!\!\Vdash .\sqsubseteq_1 .\sqsubseteq_2 .f \;=\; g$$

where

$$\begin{array}{lcll} [t]g.a.(\top_1, c) & = & \top_3 & \\ g.a.(X, c) & = & (f.a.b.X, b) & \text{, if } X \neq \top_1 \end{array}$$

where $\top_1$ belongs to $\sqsubseteq_1$ and $\top_3$ to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$.

Definition 6.3.2 $\lceil \varrho \rceil$ AND $\lceil \delta \rceil$

$$\begin{array}{lcl} \lceil \varrho \rceil & = & \rightarrowtail\!\!\Vdash .\sqsubseteq_1 .\sqsubseteq_1 .\mathsf{addW},\ (e_1, e_2) \Vdash\ \varrho \\ \lceil \delta \rceil & = & \rightarrowtail\!\!\Vdash .\sqsubseteq_1 .\sqsubseteq_1 .\mathsf{addW},\ (e_1, e_2),\ \sqsubseteq_1 \bigotimes \sqsubseteq_2 \Vdash\ \delta \end{array}$$

Theorem 6.3.3 $\rightarrowtail\!\!\Vdash$ $\sqcap$-CLOSEDNESS *CAP_Closed_LEXII*

$$\frac{\mathsf{Lattice}.\sqsubseteq_1\ \wedge \mathsf{Lattice}.\sqsubseteq_2\ \wedge\ \sqsubseteq_1\ \text{is } \sqcap_1\text{-closed} \wedge\ \sqsubseteq_2\ \text{is } \sqcap_2\text{-closed}}{\sqsubseteq_1 \bigotimes \sqsubseteq_2\ \text{ is } \sqcap_3\text{-closed}}$$

where $\sqsubseteq_3$ belongs to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$.

Theorem 6.3.4 $\rightarrowtail\!\!\Vdash$ DISTRIBUTIVE *Fish_Distr*

$$\frac{\mathsf{Lattice}.\sqsubseteq_1\ \wedge \mathsf{Lattice}.\sqsubseteq_2\ \wedge\ \sqsubseteq_1\ \text{is } \sqcap_1\text{-closed}\ \wedge\ (\forall a, b :: f.a.b \text{ is } \sqcap_1\text{-distributive})}{\rightarrowtail\!\!\Vdash .\sqsubseteq_1 .\sqsubseteq_2 .f.a.b \text{ is } \sqcap_3\text{-distributive}}$$

where $\sqcap_3$ corresponds to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$.

Theorem 6.3.5 $\rightarrowtail\!\!\Vdash$ FIRST *FST_MiCost_Fish*

$$\frac{\mathsf{Lattice}.\sqsubseteq_1\ \wedge\ \mathsf{Lattice}.\sqsubseteq_2}{\mathsf{fst}.(\lceil \delta \rceil.U.b) = \delta.U.b}$$

Theorem 6.3.6 BEST NEIGHBOR *MiCost_Fish_THM*

$$\frac{\begin{array}{c} \mathsf{Lattice}.\sqsubseteq_1\ \wedge \mathsf{Lattice}.\sqsubseteq_2 \\ \sqsubseteq_1\ \text{is } \sqcap_1\text{-closed} \wedge\ \sqsubseteq_2\ \text{is } \sqcap_2\text{-closed}\ \wedge\ (\forall a, b :: \mathsf{addW}.a.b \text{ is } \sqcap_1\text{-distributive}) \\ (\delta.U.b \neq \top_1)\ \wedge\ (U \neq \emptyset)\ \wedge\ (N.b \neq \emptyset)\ \wedge\ b \notin U \end{array}}{\delta.U.b\ =\ \mathsf{addW}.b'.b.(\delta.U.b')}$$

where $\top_1$ belongs to $\sqsubseteq_1$, $\sqcap_3$ belongs to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$, and $b' = \mathsf{snd}.(\lceil \delta \rceil.U.b)$.

Figure 6.4: *Some theorems on best neighbor function.*

6.4 Round Solvability

Theorems δ SELF$_{121}$ and δ SPLIT$_{121}$ give a recursive definition for the minimum cost function δ. This can be rewritten by introducing a higher order function φ to obtain an equation of the form:

$$\delta.U.b \;=\; \varphi.U.b.\delta$$

The function δ is thus a fix point of $(\lambda\, f, U, b.\ \varphi.U.b.f)$. One may ask whether δ is the least fix point with respect to some lattice (and hence can be approached by a successive approximation), or even stronger: whether it is the only fix point. Much more relevant to us, is the question whether δ can be approached from any point, and therefore a self-stabilizing computation will be possible. We already have half an answer to this question: the FSA algorithm is our proof, but first we need to show that δ is round solvable. The function φ —also called *generator*— plays an important role, as it is exactly the function required by the round solvability condition. Note that δ used here has a slightly different type than the one expected by the FSA algorithm: the type of δ here is $\mathcal{P}(V){\to}V{\to}A$ whereas the algorithm expects $V{\to}V{\to}A$. So, we are not going to use the algorithm to fully compute δ, but $\delta' = (\lambda a, b.\ \delta.\{a\}.b)$. We do have a good reason to assume a more general type of δ, this will be made clear later. Hopefully, this will not cause too much confusion.

Definition 6.4.1 MINIMAL COST GENERATOR *GENmc*

For any $F \in \mathcal{P}(V){\to}V{\to}A$:

$$\varphi.U.b.F \;=\; \begin{cases} e & \text{, if } b \in U \\ \sqcap\{\mathsf{addW}.b'.b.(F.b') \mid b' \in N.b\} & \text{, otherwise} \end{cases}$$

Theorem 6.4.2 *MiCost_SAT_Spec*

$$\frac{\mathsf{Lattice}.\sqsubseteq \;\wedge\; (\forall a, b :: \mathsf{addW}.a.b \text{ is } \sqcap\text{-distributive}) \;\wedge\; (e = \bot)}{\delta.U.b \;=\; \varphi.U.b.\delta}$$

---◀

In Corollary$_{111}$ 5.5.2 it was stated that the FSA algorithm will self-stabilizingly compute δ if δ satisfies the round solvability condition stated in Theorem$_{111}$ 5.5.1. The condition requires the existence of a well-founded relation $\sqsubset$ over A, a predicate $\mathsf{ok} \in A{\to}V{\to}V{\to}A{\to}\mathbb{B}$, and a function $\Phi \in V{\to}V{\to}(V{\to}A){\to}A$ satisfying:

$$(\forall m, b' : m \sqsubset n \wedge b' \in N.b :\ \mathsf{ok}^{m}_{a,b'}.(F.b')) \;\Rightarrow\; \mathsf{ok}^{n}_{a,b}.(\Phi.a.b.F)$$

for all $a, b \in V$ and $F \in V{\to}A$. The choice of $\sqsubset$, ok, and Φ has been motivated before and is summarized in Corollary$_{111}$ 5.5.2. Before we continue with verifying this condition, recall that the δ used here has a more general type. So, we will instead prove a slightly stronger property:

$$(\forall m, b' : m \sqsubset n \wedge b' \in N.b :\ \mathsf{ok}^{m}_{U,b'}.(F.b')) \;\Rightarrow\; \mathsf{ok}^{n}_{U,b}.(\Phi.U.b.F) \qquad (6.4.1)$$

The type of ok and Φ should also be lifted accordingly. The choice for ok is as according to Corollary$_{111}$ 5.5.2, but generalized to accommodate the type change:

$$\mathsf{ok}^n_{U,b}.X \;=\; (\delta.U.b \sqsubseteq n \Rightarrow (X = \delta.U.b)) \;\wedge\; (n \sqsubseteq \delta.U.b \Rightarrow n \sqsubseteq X) \tag{6.4.2}$$

The Φ suggested by the corollary is an instance of the one defined in Definition 6.4.1. The latter will be used now as it has the correct type for the above equation. For $\sqsubset$ is was proposed to take $\sqsubseteq -I$ where $\sqsubseteq$ is the relation on which the $\sqcap$ operator used in the function δ is based.

The following abbreviation is useful for later:

$$\mathsf{pre}^n_{U,b}.F \;=\; (\forall m, b' : m \sqsubset n \wedge b' \in N.b :\; \mathsf{ok}^m_{U,b'}.(F.b')) \tag{6.4.3}$$

Or, alternatively:

$$\mathsf{pre}^n_{U,b}.F \;=\; (\forall m, b' : m \sqsubset n \wedge b' \in N.b : \left\{ \begin{array}{l} \delta.U.b' \sqsubset n \Rightarrow (F.b' = \delta.U.b) \;\;\wedge \\ m \sqsubseteq \delta.U.b' \Rightarrow m \sqsubseteq F.b' \end{array} \right.) \tag{6.4.4}$$

Using pre we can rewrite (6.4.1) to the following, more compact, formula. For all $n \in A$, $U \subseteq V$, $b \in V$, and $F \in V \to A$:

$$\mathsf{pre}^n_{U,b}.F \;\Rightarrow\; \mathsf{ok}^n_{U,b}.(\varphi.U.b.F) \tag{6.4.5}$$

That is what we are going to prove now.

This is going to be the longest proof presented in this thesis. If the reader wishes, he can skip the calculation. The results are presented in Theorems$_{129}$ 6.4.3 and 6.4.4. Still, we encourage the reader to bear with the calculation. It is not really that difficult and it exposes in detail where and how various conditions required to obtain round-solvability play their role.

Assume in the sequel that $\sqsubseteq$ is a lattice which is $\sqcap$-closed.

Let $b \in V$. We distinguish two cases: $b \in U$ and $b \notin U$. For the first case we derive:

$$\begin{array}{ll}
 & \mathsf{ok}^n_{U,b}.(\varphi.U.b.F) \\
= & \quad \{ \text{ definition}_{125} \text{ of } \varphi,\ b \in U \ \} \\
 & \mathsf{ok}^n_{U,b}.e \\
= & \quad \{ \text{ definition (6.4.2) of } \mathsf{ok} \ \} \\
 & (\delta.U.b \sqsubseteq n \Rightarrow (e = \delta.U.b) \;\wedge\; (n \sqsubset \delta.U.b \Rightarrow n \sqsubseteq e) \\
= & \quad \{ \text{ let } e = \bot,\ \delta \text{ S}{\small ELF}_{121} \ \} \\
 & (\bot \sqsubseteq n \Rightarrow (\bot = \bot) \;\wedge\; (n \sqsubset \bot \Rightarrow n \sqsubseteq \bot) \\
= & \quad \{ \text{ simple calculation } \} \\
 & \mathsf{true}
\end{array}$$

So, (6.4.5) is satisfied if $e = \bot$.

Let us consider now the case if $b \notin U$. We derive:

$$\begin{array}{ll}
 & \mathsf{ok}^n_{U,b}.(\varphi.U.b.F) \\
= & \quad \{ \text{ definition (6.4.2) of } \mathsf{ok} \ \}
\end{array}$$

$$
\begin{array}{ll}
 & (\delta.U.b \sqsubseteq n \Rightarrow (\varphi.U.b.F = \delta.U.b)) \ \wedge \ (n \sqsubseteq \delta.U.b \Rightarrow n \sqsubseteq \varphi.U.b.F) \\
= & \quad \{ \ b \notin U, \text{ definition}_{125} \text{ of } \varphi \ \} \\
 & (\delta.U.b \sqsubseteq n \Rightarrow (\sqcap\{\mathsf{addW}.b'.b.(F.b') \mid b' \in N.b\} = \delta.U.b)) \ \wedge \\
 & (n \sqsubseteq \delta.U.b \Rightarrow n \sqsubseteq \sqcap\{\mathsf{addW}.b'.b.(F.b') \mid b' \in N.b\}) \\
\Leftarrow & \quad \{ \text{ definition of } \sqcap \ \} \\
 & (\delta.U.b \sqsubseteq n \ \Rightarrow \ (\exists a : a \in N.b : \mathsf{addW}.a.b.(F.a) = \delta.U.b) \quad \wedge \\
 & \qquad\qquad\qquad (\forall b' : b' \in N.b : \delta.U.b \sqsubseteq \mathsf{addW}.b'.b.(F.b'))) \quad \wedge \\
 & (n \sqsubseteq \delta.U.b \ \Rightarrow \ (\forall b' : b' \in N.b : n \sqsubseteq \mathsf{addW}.b'.b.(F.b'))) \\
\Leftarrow & \quad \{ \ \sqsubseteq \text{ is } \sqcap\text{-closed, } (6.1.2)_{115} \ \} \\
 & (\delta.U.b \sqsubseteq n \ \Rightarrow \ (\exists a : a \in N.b : \mathsf{addW}.a.b.(F.a) = \delta.U.b) \quad \wedge \\
 & \qquad\qquad\qquad (\forall b' : b' \in N.b : \delta.U.b \sqsubseteq \mathsf{addW}.b'.b.(F.b'))) \quad \wedge \\
 & (n \sqsubseteq \delta.U.b \ \Rightarrow \ (\forall m, b' : b' \in N.b \wedge m \sqsubset n : m \sqsubset \mathsf{addW}.b'.b.(F.b')))
\end{array}
$$

So, it suffices to show:

$$\mathsf{pre}^n_{U,b}.F \ \wedge \ \delta.U.b \sqsubseteq n \ \Rightarrow \ \mathsf{addW}.a.b.(F.a) = \delta.U.b) \tag{6.4.6}$$

$$\mathsf{pre}^n_{U,b}.F \ \wedge \ \delta.U.b \sqsubseteq n \ \Rightarrow \ \delta.U.b \sqsubseteq \mathsf{addW}.b'.b.(F.b') \tag{6.4.7}$$

$$\mathsf{pre}^n_{U,b}.F \ \wedge \ n \sqsubseteq \delta.U.b \ \wedge \ m \sqsubset n \ \Rightarrow \ m \sqsubset \mathsf{addW}.b'.b.(F.b') \tag{6.4.8}$$

for all $n \in A$, $U \subseteq V$, $b \in V$, $b' \in N.b$, $F \in V \rightarrow A$, $m \in A$, and *some* $a \in N.b$[12].

For 6.4.6 we have to show the existence of an a satisfying the equation. First we derive:

$$
\begin{array}{ll}
 & \mathsf{addW}.a.b.(F.a) = \delta.U.b \\
\Leftarrow & \quad \{ \text{ simple calculation } \} \\
 & (\mathsf{addW}.a.b.(\delta.U.a) = \delta.U.b) \ \wedge \ (F.a = \delta.U.a) \\
\Leftarrow & \quad \{ \ \mathsf{pre}^n_{U,b} \text{ is assumed, definition (6.4.4) of } \mathsf{pre}^n_{U,b} \ \} \\
 & (\mathsf{addW}.a.b.(\delta.U.a) = \delta.U.b) \ \wedge \ (\delta.U.a \sqsubset n) \ \wedge \ a \in N.b \\
\Leftarrow & \quad \{ \ \delta.U.b \sqsubseteq n \text{ is assumed, } \sqsubseteq \text{ is transitive } \} \\
 & (\mathsf{addW}.a.b.(\delta.U.a) = \delta.U.b) \ \wedge \ (\delta.U.a \sqsubset \delta.U.b) \ \wedge \ a \in N.b \\
= & \quad \{ \ (\dagger) \text{ insist that } (x \neq \top) \ \Rightarrow \ x \sqsubseteq \mathsf{addW}.a.b.x \ \} \\
 & (\mathsf{addW}.a.b.(\delta.U.a) = \delta.U.b) \ \wedge \ a \in N.b \ \wedge \ (\delta.U.a \neq \top)
\end{array}
$$

[12] Note that this requires that $N.b \neq \emptyset$. So, what if $N.b$ is empty? If this is the case, notice that, by definition of φ we have $\varphi.U.b.F = \top$. So:

$$
\begin{array}{ll}
 & \mathsf{ok}^n_{U,b}.(\varphi.U.b.F) \\
= & \quad \{ \text{ definition (6.4.2) of } \mathsf{ok} \ \} \\
 & (\delta.U.b \sqsubseteq n \Rightarrow (\varphi.U.b.F = \delta.U.b)) \ \wedge \ (n \sqsubseteq \delta.U.b \Rightarrow n \sqsubseteq \varphi.U.b.F) \\
= & \quad \{ \text{ the remark above } \} \\
 & (\delta.U.b \sqsubseteq n \Rightarrow (\top = \delta.U.b)) \ \wedge \ (n \sqsubseteq \delta.U.b \Rightarrow n \sqsubseteq \top)
\end{array}
$$

which is satisfies if $\delta.U.b = \top$. Since $N.b = \emptyset$ and $b \notin U$, there exists no path from U to b and hence by the definition of δ, $\delta.U.b$ is indeed equal to $\top$.

So, (6.4.6) can be satisfied by taking the best neighbor of b as a. The existence of such a neighbor is guaranteed by Theorem BEST NEIGHBOR EXISTENCE$_{121}$. Notice also the requirement $(x \neq \top)\ x \sqsubseteq \mathsf{addW}.a.b.x$ in (†). This is the non-zero link-cost condition we mentioned earlier in (6.2.5).

For (6.4.7) we distinguish two cases: $\delta.U.b \sqsubseteq \delta.U.b'$ and $\delta.U.b \not\sqsubseteq \delta.U.b'$. For the first case we derive:

$$\begin{array}{ll}
& \delta.U.b \sqsubseteq \mathsf{addW}.b'.b.(F.b') \\
= & \{ \text{ insists } \mathsf{addW}.b'.b \text{ is } \sqcap\text{-distributive, hence } \mathsf{addW}.b'.b.\top = \top \ \} \\
& (F.b' \neq \top) \ \Rightarrow\ \delta.U.b \sqsubseteq \mathsf{addW}.b'.b.(F.b') \\
= & \{ \sqsubseteq \text{ is } \sqcap\text{-closed, } (6.1.2)_{115} \ \} \\
& (F.b' \neq \top) \ \Rightarrow\ (\forall m : m \sqsubset \delta.U.b : m \sqsubset \mathsf{addW}.b'.b.(F.b')) \\
\Leftarrow & \{ \text{ insist that } (x \neq \top) \ \Rightarrow\ x \sqsubset \mathsf{addW}.a.b.x \ \} \\
& (\forall m : m \sqsubset \delta.U.b : m \sqsubseteq F.b') \\
\Leftarrow & \{ \ \mathsf{pre}_b^n \text{ is assumed, definition (6.4.4) of } \mathsf{pre}_b^n \ \} \\
& (\forall m : m \sqsubset \delta.U.b : m \sqsubset n \wedge m \sqsubseteq \delta.U.b') \\
= & \{ \sqsubseteq \text{ is transitive } \} \\
& \delta.U.b \sqsubseteq n \ \wedge\ \delta.U.b \sqsubseteq \delta.U.b'
\end{array}$$

So, (6.4.7) is satisfied if $\delta.U.b \sqsubseteq \delta.U.b'$. Now for the second case:

$$\begin{array}{ll}
& \delta.U.b \sqsubseteq \mathsf{addW}.b'.b.(F.b') \\
\Leftarrow & \{ \ \mathsf{addW}.b'.b \text{ is } \sqcap\text{-distributive, hence monotonic } \} \\
& \delta.U.b \sqsubseteq \mathsf{addW}.b'.b.(\delta.U.b') \ \wedge\ \delta.U.b' \sqsubseteq F.b' \\
= & \{ \ b' \in N.b,\ \delta \text{ LESS}_{121} \ \} \\
& \delta.U.b' \sqsubseteq F.b' \\
\Leftarrow & \{ \ \mathsf{pre}_b^n \text{ is assumed, definition (6.4.4) of } \mathsf{pre}_b^n \ \} \\
& \delta.U.b' \sqsubset n \ \wedge\ \delta.U.b' \sqsubseteq \delta.U.b' \\
\Leftarrow & \{ \sqsubseteq \text{ is reflexive and transitive } \} \\
& \delta.U.b' \sqsubset \delta.U.b \ \wedge\ \delta.U.b \sqsubseteq n \\
= & \{ \sqsubseteq \text{ is } \sqcap\text{-closed, } (6.1.1)_{115} \ \} \\
& \delta.U.b \not\sqsubseteq \delta.U.b' \ \wedge\ \delta.U.b \sqsubseteq n
\end{array}$$

From the above calculation, and the previous one, we conclude that (6.4.7) too is satisfied, provided the non-zero link-cost property holds.

For (6.4.8) we derive:

$$\begin{array}{ll}
& m \sqsubseteq \mathsf{addW}.b'.b.(F.b') \\
= & \{ \text{ insists } \mathsf{addW}.b'.b \text{ is } \sqcap\text{-distributive, hence } \mathsf{addW}.b'.b.\top = \top \ \} \\
& (F.b' \neq \top) \ \Rightarrow\ m \sqsubseteq \mathsf{addW}.b'.b.(F.b') \\
\Leftarrow & \{ \text{ simple calculation, insist } (x \neq \top) \ \Rightarrow\ x \sqsubset \mathsf{addW}.b'.b.x \ \}
\end{array}$$

$$
\begin{array}{ll}
 & (m \sqsubset \mathsf{addW}.b'.b.(\delta.U.b') \ \wedge \ (F.b' = \delta.U.b')) \ \vee \ m \sqsubseteq F.b' \\
\Leftarrow & \{ \ \mathsf{pre}^n_b \text{ is assumed, definition (6.4.4) of } \mathsf{pre}^n_b \ \} \\
 & (m \sqsubset \mathsf{addW}.b'.b.(\delta.U.b') \ \wedge \ \delta.U.b' \sqsubset n) \ \vee \ m \sqsubseteq \delta.U.b' \\
\Leftarrow & \{ \ \sqsubseteq \text{ is transitive } \} \\
 & (m \sqsubset n \ \wedge \ n \sqsubseteq \delta.U.b \ \wedge \ \delta.U.b \sqsubseteq \mathsf{addW}.b'.b.(\delta.U.b') \ \wedge \ \delta.U.b' \sqsubseteq m) \ \vee \ m \sqsubseteq \delta.U.b' \\
\Leftarrow & \{ \ b' \in N.b, \ \delta \ \textsc{Less}_{121}, \text{ simple calculation } \} \\
 & (m \sqsubset n \ \wedge \ n \sqsubseteq \delta.U.b) \ \wedge \ (\delta.U.b' \sqsubseteq m \ \vee \ m \sqsubseteq \delta.U.b') \\
= & \{ \ (6.1.1)_{115}, \text{ some calculation } \} \\
 & m \sqsubset n \ \wedge \ n \sqsubseteq \delta.U.b
\end{array}
$$

So, (6.4.8) is also satisfied.

We conclude therefore that (6.4.5) is satisfied. If $\sqsubset$ is also well-founded[13], then it follows that the round-solvability condition$_{126}$ 6.4.2 is also satisfied. Hence, $\delta' = (\lambda a, b.\ \delta.\{a\}.b)$ can be computed self-stabilizingly by the FSA algorithm. Notice also how the $\sqcap$-closedness of $\sqsubseteq$ and the non-zero link-cost condition, the latter is:

$$(x \neq \top) \ \Rightarrow \ x \sqsubset \mathsf{addW}.b'.b.x$$

are repeatedly invoked, a fact which underscores the role of these two conditions.

Theorem 6.4.3 below re-states the result of the calculation. Since a best neighbor function $\lceil\delta\rceil$ —see Section 6.3— is also a minimal cost function$_{124}$ (Definition 6.3.2), the round solvability result of Theorem 6.4.3 also applies to $\lceil\delta\rceil$. This is re-stated by Theorem 6.4.4.

Theorem 6.4.3 δ Round Solvability *MiCost_Round_Solv*

$$
\frac{\begin{array}{c}\mathsf{Lattice}.\sqsubseteq \ \wedge \ \sqsubseteq \text{ is } \sqcap\text{-closed} \\ (\forall a, b :: \mathsf{addW}.a.b \text{ is } \sqcap\text{-distributive}) \ \wedge \ (\forall a, b, x : x \neq \top : x \sqsubset \mathsf{addW}.a.b.x) \\ (e = \bot) \ \wedge \ (\delta.U.b \neq \top) \ \wedge \ (U \neq \emptyset)\end{array}}{(\forall b', m : b' \in N.b \wedge m \sqsubset n : \mathsf{ok}^n_{U,b}.(F.b')) \ \Rightarrow \ \mathsf{ok}^n_{U,b}.(\varphi.U.b.F)}
$$

where $\mathsf{ok}^n_{U,b}.X$ is as in (6.4.2).

Theorem 6.4.4 $\lceil\delta\rceil$ Round Solvability *MiCostFish_Round_Solv*

$$
\frac{\begin{array}{c}\mathsf{Lattice}.\sqsubseteq_1 \ \wedge \ \mathsf{Lattice}.\sqsubseteq_2 \ \wedge \ \sqsubseteq_1 \text{ is } \sqcap_1\text{-closed} \ \wedge \ \sqsubseteq_2 \text{ is } \sqcap_2\text{-closed} \\ (\forall a, b :: \mathsf{addW}.a.b \text{ is } \sqcap_1\text{-distributive}) \ \wedge \ (\forall a, b, x : x \neq \top_1 : x \sqsubset_1 \mathsf{addW}.a.b.x) \\ (e_1, e_2 = \bot_1, \bot_2) \ \wedge \ (\delta.U.b \neq \top) \ \wedge \ (U \neq \emptyset)\end{array}}{\begin{array}{c} \succ\!\!\Vdash . \sqsubseteq_1 . \sqsubseteq_2 . \mathsf{addW}, \ (e_1, e_2), \ \sqsubseteq_3 \\ \Vdash \\ (\forall b', m : b' \in N.b \wedge m \sqsubset_3 n : \mathsf{ok}^n_{U,b}.(F.b')) \ \Rightarrow \ \mathsf{ok}^n_{U,b}.(\varphi.U.b.F)\end{array}}
$$

where $\sqsubseteq_3 = \sqsubseteq_1 \bigotimes \sqsubseteq_2$ and $\mathsf{ok}^n_{U,b}.X$ is as in (6.4.2).

---

[13] Notice that the calculation requires $\sqsubseteq$ to be a $\sqcap$-closed lattice, and hence $\sqsubset$ is well-founded.

6.4.1 The Fix Point of a Generator

Basically, the FSA algorithm performs a successive and fair approximation using the generator φ, or perhaps one should say, a set of generators $\varphi.U.b$'s. The fact that the algorithm is self-stabilizing means that the order in which the generators are applied at each step is irrelevant, and that the process converges to the corresponding minimal cost function δ regardless of its initial state. Theorem$_{125}$ 6.4.2 states that δ is also a fix point of φ. It is also the *only* fix point. If another fix point δ' exists, we can start the FSA algorithm with $d = \delta'$[14]. Since it is a fix point, any application of a generator will leave d to remain equal to δ'. However, we have argued that the algorithm will always converge to $d = \delta$. Hence $\delta = \delta'$.

Round solvability is actually a very strong condition. The above argument on the uniqueness of the fix point of the generator φ extends to any generator satisfying the round solvability condition. In addition, under some reasonable condition, the round solvability implies the existence of the fix point itself.

Let $\Phi \in \mathcal{P}(V) \to V \to (V \to A) \to A$ be a generator satisfying (6.4.1) for *some* predicate ok and some well-founded relation $\sqsubset$. That is:

$$(\forall m, b' : b' \in N.b \wedge m \sqsubset n : \mathsf{ok}^n_{U,b'}.(F.b')) \;\Rightarrow\; \mathsf{ok}^n_{U,b}.(\Phi.U.b.F) \qquad (6.4.9)$$

for all $U \subseteq V$, $b \in V$, $n \in A$, $F \in V \to A$. By Theorem$_{111}$ 5.5.1 the FSA algorithm will satisfy:

$$\mathsf{true} \vdash \mathsf{true} \rightsquigarrow (\forall n, a, b : n \in A \wedge a, b \in V : \mathsf{ok}^n_{\{a\},b}.(d.a.b)) \qquad (6.4.10)$$

Let the goal be to compute Δ, or more precisely, to converge to $(\forall a, b : a, b \in V : d.a.b = \Delta.\{a\}.b)$. By the convergence property above, it suffices if ok is such that:

$$\mathsf{ok}^{\Delta.U.b}_{U,b}.X \;\Rightarrow\; (X = \Delta.U.b) \qquad (6.4.11)$$

assuming that $(\forall U, b : U \subseteq V \wedge b \in V : \Delta.U.b \in A)$. In terms of FSA algorithm, the above will mean that it takes at most $\Delta.\{a\}.b$ rounds for the algorithm to compute the right $d.a.b$. The above seems to be a reasonable requirement. Notice that the minimal cost generator φ (Definition$_{125}$ 6.4.1) also satisfies the above.

In addition, $\mathsf{ok}^n_{U,b}.X$ in intended to express when the value X is considered an acceptable approximation of $\Delta.U.b$ at round n. Obviously, if X is $\Delta.U.b$ itself, then $\mathsf{ok}^n_{U,b}.X$ should be true. That is, ok has to be such that the following holds:

$$(\forall n, U, b : n \in A \wedge U \subseteq A \wedge b \in V : \mathsf{ok}^n_{U,b}.(\Delta.U.b)) \qquad (6.4.12)$$

If we have the generator Φ and the predicate ok are such that (6.4.9), (6.4.11), and (6.4.12) are satisfied then we can show that Δ is a fix point of Φ:

$$\begin{array}{ll} & (\forall n : n \in A : \mathsf{ok}^n_{U,b}.(\Delta.U.b)) \\ \Rightarrow & \{ \text{ simple calculation } \} \end{array}$$

[14] In addition —a minor detail—, assume that the link-registers do not contain 'bad' values

$$
\begin{array}{ll}
 & (\forall m, b' : b' \in N.b \wedge m \sqsubset \Delta.U.b : \mathsf{ok}^m_{U,b'}.(\Delta.U.b)) \\
\Rightarrow & \{\ (6.4.9)\ \} \\
 & \mathsf{ok}^{\Delta.U.b}_{U,b}.(\Phi.U.b.\Delta) \\
\Rightarrow & \{\ (6.4.11)\ \} \\
 & \Delta.U.b = \Phi.U.b.\Delta
\end{array}
$$

Hence we conclude that Δ is a fix point of Φ:

$$(\forall U, b : U \subseteq A \wedge b \in V : \Delta.U.b = \varphi.U.b.\Delta) \tag{6.4.13}$$

The uniqueness of Δ can be demonstrated as follows. Let ∇ be a fix point of Φ. That is, ∇ satisfies equation (6.4.13). We derive:

$$
\begin{array}{ll}
 & \Delta.U.b = \nabla.U.b \\
\Leftarrow & \{\ (6.4.11)\ \} \\
 & \mathsf{ok}^{\Delta.U.b}_{U,b}.(\nabla.U.b) \\
\Leftarrow & \{\ \text{simple calculation}\ \} \\
 & (\forall n, b : n \in A \wedge b \in V : \mathsf{ok}^n_{U,b}.(\nabla.U.b)) \\
\Leftarrow & \{\ \sqsubset \text{ is well-founded, well-founded induction}\ \} \\
 & (\forall n, b : b \in V : (\forall m, b' : m \sqsubset n \wedge b' \in V : \mathsf{ok}^m_{U,b'}.(\nabla.U.b')) \ \Rightarrow\ \mathsf{ok}^n_{U,b}.(\nabla.U.b)) \\
= & \{\ \nabla \text{ is a fix point of } \Phi\ \} \\
 & (\forall n, b : b \in V : (\forall m, b' : m \sqsubset n \wedge b' \in V : \mathsf{ok}^m_{U,b'}.(\nabla.U.b')) \ \Rightarrow\ \mathsf{ok}^n_{U,b}.(\Phi.U.b.\nabla)) \\
\Leftarrow & \{\ \text{predicate calculus}\ \} \\
 & (\forall n, b : b \in V : (\forall m, b' : m \sqsubset n \wedge b' \in N.b : \mathsf{ok}^m_{U,b'}.(\nabla.U.b')) \Rightarrow \mathsf{ok}^n_{U,b}.(\Phi.U.b.\nabla)) \\
= & \{\ (6.4.9)\ \} \\
 & \mathsf{true}
\end{array}
$$

So, we conclude that Δ and ∇ are equal, or more precisely:

$$(\forall a, b : U \subseteq V \wedge b \in V : \Delta.U.b = \nabla.U.b) \tag{6.4.14}$$

The fact that a round-solvable function is a fix point of its generator and that this fix point is unique, is stated more precisely in the theorems in Figure 6.5.

6.5 Self-Stabilizing Broadcast

Recall that the function $\lceil\delta\rceil$ returns not only the minimal cost between a source U and a destination b, but also the best neighbor of b. Together, the best neighbors define the best paths from U to any node in the network. This information can be used to broadcast data from U across the network by simply following the paths induced by the best neighbors. Note that we do not actually have to record the best neighbors since the minimum cost to the neighbors of a node defines the node's best neighbors. Recall that $\lceil\delta\rceil$ is obtained by replacing the lattice $\sqsubseteq_1$ and the function addW used

Theorem 6.4.5 GENERATOR'S FIX POINT *FP_Gen*
Let $f \in V \to A$, $\sqsubset$ be a relation over A, $\mathsf{ok} \in A \to V \to A \to \mathbb{B}$, and $\Phi \in V \to (V \to A) \to A$. We have:

$$\frac{\begin{array}{c}(\forall n, b, g : b \in V : (\forall m, b' : m \sqsubset n \wedge b' \in N.b : \mathsf{ok}^m_{b'}.(g.b')) \Rightarrow \mathsf{ok}^n_b.(\Phi.b.g)) \\ (\forall b, x : b \in V : \mathsf{ok}^{f.b}_b.x \Rightarrow (x = f.b)) \;\wedge\; (\forall n, b : b \in V : \mathsf{ok}^n_b.(f.b))\end{array}}{(\forall b : b \in V : f.b = \Phi.b.f)}$$

Theorem 6.4.6 GENERATOR'S UNIQUE FIX POINT *FP_Gen_UNIQUE*
Let (V, N) be a network, and let f, $\sqsubset$, ok, and Φ have the types as in Theorem 6.4.5. In addition, $\sqsubset$ is well-founded. We have:

$$\frac{\begin{array}{c}(\forall n, b, g : b \in V : (\forall m, b' : m \sqsubset n \wedge b' \in N.b : \mathsf{ok}^m_{b'}.(g.b')) \Rightarrow \mathsf{ok}^n_b.(\Phi.b.g)) \\ (\forall b, x : b \in V : \mathsf{ok}^{f.b}_b.x \Rightarrow (x = f_1.b)) \;\wedge\; (\forall b : b \in V : f_2.b = \Phi.b.f_2)\end{array}}{(\forall b : b \in V : f_1.b = f_2.b)}$$

Figure 6.5: *Theorems about the Fix Point of a Generator.*

in the minimal cost function with the lexicographic product $\sqsubseteq_1 \bigotimes \sqsubseteq_2$ and $\succ\!\!\!+\!\!\!+\!.\mathsf{addW}$. We can apply a similar trick, namely by replacing $\sqsubseteq_1$ with $\sqsubseteq_1 \bigotimes \sqsubseteq_2$, and addW with $\succ\!\!\!-\!\!\!\prec.\mathsf{addW}$ where $\succ\!\!\!-\!\!\!\prec$, we call it *crab* operator, is defined as follows:

Definition 6.5.1 CRAB OPERATOR *Crab*
For all $f \in V \to V \to A \to A$:

$$\succ\!\!\!-\!\!\!\prec.\sqsubseteq_1.\sqsubseteq_2.f \;=\; g$$

where

$$\begin{array}{lcll} g.a.b.(\top_1, Y) & = & \top_3 & \\ g.a.b.(X, Y) & = & (f.a.b.X, Y) & \text{, if } X \neq \top_1 \end{array}$$

where $\top_3$ belongs to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$.

So, given an 'add link-weight function' addW, the function $\succ\!\!\!-\!\!\!\prec.\sqsubseteq_1.\sqsubseteq_2.\mathsf{addW}.a.b$, when applied to a pair (X, Y) will simply pass Y to the second component of its result (this corresponds with the message Y being passed from the node a to the node b in the actual broadcast process), whereas the result of applying addW will be returned in the first component of the result.

Let the corresponding path-cost and minimal cost functions be denoted by $\langle \varrho \rangle$ and $\langle \delta \rangle$:

$$\langle \varrho \rangle_g.b \;=\; \succ\!\!\!-\!\!\!\prec.\sqsubseteq_1.\sqsubseteq_2.\mathsf{addW},\; (e_1, g.b) \Vdash \varrho.b \qquad (6.5.1)$$

$$\langle \delta \rangle_g.U.b \;=\; \succ\!\!\!-\!\!\!\prec.\sqsubseteq_1.\sqsubseteq_2.\mathsf{addW},\; (e_1, g.b),\; \sqsubseteq_1 \bigotimes \sqsubseteq_2 \Vdash \delta.U.b \qquad (6.5.2)$$

The function g is intended to contain the data which are to be broadcasted. U contains the source-nodes of the broadcast, and $g.b$ is the datum which belongs to the source-node b. These source data may all be identical, but we do not require them to be so. We will return to this issue later.

The path-cost and minimal cost functions defined above have the following properties:

$$\begin{aligned} \mathsf{fst}.(\langle\varrho\rangle_g.b.s) &= \mathsf{addW},\ e_1 \Vdash \varrho.b.s && (6.5.3)\\ \mathsf{snd}.(\langle\varrho\rangle_g.b.[]) &= g.b && (6.5.4)\\ \mathsf{snd}.(\langle\varrho\rangle_g.b.(s;a)) &= g.a \qquad \text{, if } \mathsf{fst}.(\langle\varrho\rangle_g.b.s) \neq \top_1 && (6.5.5)\\ \mathsf{fst}.(\langle\delta\rangle_g.U.b) &= \mathsf{addW},\ e_1,\ \sqsubseteq_1 \Vdash \delta.U.b && (6.5.6) \end{aligned}$$

Notice that the new path-cost and minimal cost functions return in their first components the same values as the original functions. In the sequel we will denote $\mathsf{addW}, e_1 \Vdash \varrho.b.s$ simply by $\varrho.b.s$, and $\mathsf{addW}, e_1, \sqsubseteq_1 \Vdash \delta.U.b$ by $\delta.U.b$.

If $\langle\delta\rangle$ is really a broadcast function, we expect that evaluating $\mathsf{snd}.(\langle\delta\rangle_g.U.b)$ will return a datum from a source node, that is, $g.a$ for some $a \in U$. Let us see if this is the case. Let $\sqcap_3$ be the greatest upper bound operator that belongs to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$. We derive:

$$\begin{aligned}
& \mathsf{snd}.(\langle\delta\rangle_g.U.b)\\
= & \quad \{\ (6.5.2),\ \text{definition}_{119} \text{ of } \delta\ \}\\
& \mathsf{snd}.(\sqcap_3\{\langle\varrho\rangle_g.b.s \mid U \xrightarrow{s} b\})\\
= & \quad \{\ \text{definition } \sqcap_3,\ (6.1.4)_{116}\ \}\\
& \sqcap_2\{\mathsf{snd}.(\langle\varrho\rangle_g.b.s) \mid U \xrightarrow{s} b\ \wedge\ \mathsf{fst}.(\langle\varrho\rangle_g.b.s) = \sqcap_1\{\mathsf{fst}.(\langle\varrho\rangle_g.b'.t) \mid U \xrightarrow{t} b'\}\}\\
= & \quad \{\ (6.5.3),\ \text{definition}_{119} \text{ of } \delta\ \}\\
& \sqcap_2\{\mathsf{snd}.(\langle\varrho\rangle_g.b.s) \mid U \xrightarrow{s} b \wedge \varrho.b.s = \delta.U.b\}\\
= & \quad \{\ (\dagger)\ \text{case analysis, } (6.5.4),\ \text{assume } \delta.U.b \neq \top_1, \text{ and } (6.5.5)\ \}\\
& \sqcap_2(\{g.b \mid U \xrightarrow{[]} b \wedge \varrho.b.[] = \delta.U.b\} \cup \{g.a \mid U \xrightarrow{s;a} b \wedge \varrho.b.(s;a) = \delta.U.b\})\\
= & \quad \{\ (6.2.2)_{117},\ (6.2.2)_{118},\ \text{and } (6.2.3)_{118}\ \}\\
& \sqcap_2(\{g.b \mid b \in U \wedge e_1 = \delta.U.b\} \cup \{g.a \mid U \xrightarrow{s;a} b \wedge a \in U \wedge \varrho.b.(s;a) = \delta.U.b\})
\end{aligned}$$

If the argument of $\sqcap_2$ in the last formula above is not an empty set then the formula will yield $g.a$ for some $a \in U$, and hence we have proven our conjecture above. Note that the argument is non-empty if there exists a path from U to b and its cost is exactly equal to $\delta.U.b$. The latter is met if the lattice $\sqsubseteq_1$ is $\sqcap_1$-closed. Notice also that in ($\dagger$) it is required that $\delta.U.b$ should be less than $\top$.

The next question is whether this broadcast function $\langle\delta\rangle_g$ is round solvable. If it is, then it can be computed self-stabilizingly using the FSA algorithm. For a minimum cost function, Theorem$_{129}$ 6.4.3 states that such a function is round solvable. $\langle\delta\rangle_g$ is such a function, but the theorem is rather useless in this case. It requires that $e = \bot$, or in this case $(e_1, g.b) = (\bot_1, \bot_2)$ for all b, which means that the theorem can only

Definition 6.5.2 BROADCAST GENERATOR *GENbc*

Let $g, F \in V \to A$:

$$\langle \Phi \rangle_g.U.b.F \;=\; \begin{cases} (e_1, g.b) & \text{, if } b \in U \\ \sqcap_3\{ \rhd\!\!\lhd . \sqsubseteq_1 . \sqsubseteq_2 .\mathsf{addW}.b'.b.(F.b') \mid b' \in N.b\} & \text{, otherwise} \end{cases}$$

where $\sqcap_3$ belongs to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$.

Definition 6.5.3 *OKbc*

$$\mathsf{okBC}^n_{U,b}.P.(X,Y) \;=\; \mathsf{ok}^n_{U,b}.X \;\wedge\; (\delta.U.b \sqsubset_1 n \Rightarrow P.Y)$$

where ok is as in (6.4.2) and δ as in (6.2.4).

Theorem 6.5.4 ROUND SOLVABILITY OF BROADCAST *BC_Round_Solv*

$$\frac{\begin{array}{c} \mathsf{Lattice}.\sqsubseteq_1 \;\wedge\; \mathsf{Lattice}.\sqsubseteq_2 \\ \sqsubseteq_1 \text{ is } \sqcap_1\text{-closed} \;\wedge\; \sqsubseteq_2 \text{ is } \sqcap_2\text{-closed} \\ (\forall a, b :: \mathsf{addW}.a.b \text{ is } \sqcap_1\text{-distributive}) \;\wedge\; (\forall a, b, x : x \neq \top_1 : x \sqsubset_1 \mathsf{addW}.a.b.x) \\ (\forall a : a \in U : P.(g.a)) \;\wedge\; (e_1 = \bot_1) \;\wedge\; (\delta.U.b \neq \top_1) \;\wedge\; (U \neq \emptyset) \end{array}}{(\forall b', m : b' \in N.b \wedge m \sqsubset_1 n : \mathsf{okBC}^m_{U,b'}.P.(F.b')) \;\Rightarrow\; \mathsf{okBC}^n_{U,b}.P.(\langle \Phi \rangle_g.U.b.F)}$$

Figure 6.6: *The round-solvability of broadcast functions.*

guarantee the self-stabilization property of the broadcast if the value which is being broadcasted is $\bot_2$. Obviously, this is not a very useful fact. So, we do not have much choice but to do a new proof for $\langle \delta \rangle_g$, although a large part of it will benefit from intermediate results of the calculation of Theorem$_{129}$ 6.4.3[15]. But, let us worry less about this. We are not going to present the proof anyway —the result is, of course, verified.

Usually a broadcast origins from a single node. However sometimes an application requires a broadcast from multiple sources. The hierarchical version of the FSA algorithm is an example thereof. In a multiple sources broadcast, usually the requirement is only to make all copies maintained by all nodes in the network to satisfy some given predicate P. All data $g.a$ in the source nodes a are assumed to satisfy P. Therefore, although the $g.a$'s are not necessarily identical, all their copies will satisfy P. We have defined minimal cost functions to have the type $\mathcal{P}(V) \to V \to A$ instead of simply $V \to V \to A$, so we can have multiple sources. The reason behind this is now clear.

Without proof, in Figure 6.6 we present a theorem stating the round solvability of a broadcast function.

Recall that the FSA algorithm (Figure 5.9) consists of parallel components $\mathsf{FSA}.a$.

[15] Alternatively, one can also try to generalize Theorem$_{129}$ 6.4.3

```
BC.$U$ = $([\![ b : b \in V : \mathsf{BC}.U.b)$ where BC.$U.b$ is defined as:

prog    BC.$U.b$
read    $\{r.b'.b \mid b' \in V\} \cup \{r.b.c \mid c \in V\} \cup \{g.b \mid b \in U\} \cup \{d.b\}$
write   $\{r.b.c \mid c \in V\} \cup \{d.b\}$
init    true
assign
        $d.b := \begin{cases} (e_1, g.b) & \text{, if } b \in U \\ \sqcap_3\{\succ\!\!\prec.\mathsf{addW}.b'.b.(r.b.b') \mid b' \in N.b\} & \text{, otherwise} \end{cases}$
[]      $([\![ c : b \in N.c : r.c.b := d.b)$

where $\sqcap_3$ belongs to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$.
```

Figure 6.7: *A self-stabilizing broadcast algorithm.*

Theorem$_{129}$ 6.4.3 gives a convergence property we can expect from the program. A component level variant of this theorem can easily be obtained from the calculation in Section 5.5. More precisely, if there exists a well-founded relation $\prec$ over A, a predicate $\mathsf{ok} \in A \to V \to B \to \mathbb{B}$, and generator $\Phi \in V \to (V \to B) \to B$ such that:

$$(\forall m, b' : m \prec n \wedge b' \in N.b : \mathsf{ok}^m_{b'}.(F.b')) \Rightarrow \mathsf{ok}^n_b.(\Phi.b.F)$$

then the component program FSA.a in Figure 5.9 satisfies[16]:

$$\mathsf{true} \vdash \mathsf{true} \rightsquigarrow (\forall n, b : n \in A \wedge b \in V : \mathsf{ok}^n_b.(d.a.b))$$

Combining the above and Theorem 6.5.4 we conclude that, using $\langle\Phi\rangle_g.U$ as the generator, instantiating $\prec$ with $\sqsubseteq_1$, and ok^n_b with $\mathsf{okBC}^n_{U,b}.P$, the program FSA.a, $a \in V$, will converge to:

$$(\forall n, b : n \in A \wedge b \in V : \mathsf{okBC}^n_{U,b}.P.(d.a.b))$$

if $(\forall a : a \in U : P.(g.a))$ holds. Let us do some calculation to see what the above means.

$$
\begin{array}{ll}
 & (\forall n : n \in A : \mathsf{okBC}^n_{U,b}.P.(d.a.b)) \\
\Rightarrow & \{ \ \delta.U.b \in A \text{ if } U \subseteq V \text{ and } b \in V \text{, simple calculation } \} \\
 & \mathsf{okBC}^{\delta.U.b}_{U,b}.P.(d.a.b) \\
\Rightarrow & \{ \text{ definition of } \mathsf{okBC} \ \} \\
 & P.(d.a.b)
\end{array}
$$

In other words, the $d.a.b$'s of all b in the network now satisfy P, which is exactly as intended.

Figure 6.7 displays a program which is essentially equal to FSA.a. The name is replaced, and its variables are renamed. As the conclusion of the above argumentation, the program self-stabilizingly broadcasts data from nodes in U to the rest of the

[16] All a indices are now dropped since the a is now fixed.

Let (V, N) be a network with non-empty and finite V. Let A be a finite domain of rounds and C be a domain of the data which we want to broadcast. Let $\sqsubseteq_1 \in A \rightarrow A \rightarrow \mathbb{B}$ and $\sqsubseteq_2 \in B \rightarrow B \rightarrow \mathbb{B}$, $\mathsf{addW} \in V \rightarrow V \rightarrow A \rightarrow A$, and $e_1 \in A$ satisfy:

i. $\mathsf{Lattice}.\sqsubseteq_1 \ \wedge\ \mathsf{Lattice}.\sqsubseteq_2$

ii. $\sqsubseteq_1$ is $\sqcap_1$-closed $\wedge$ $\sqsubseteq_2$ is $\sqcap_2$-closed

iii. $(\forall a, b :: \mathsf{addW}.a.b$ is $\sqcap_1$-distributive$)$ and $(\forall b : b \in V : \delta.U.b \neq \top_1)$

iv. $(\forall a, b, x : x \neq \top_1 : x \sqsubset_1 \mathsf{addW}.a.b.x)$

v. $(\forall s, a : a \in U : J.s \Rightarrow (P \circ \mathsf{snd} \circ s \circ g).a)$

vi. $(e_1 = \bot_1) \ \wedge\ (U \neq \emptyset)$

for some given $U \subseteq V$ and predicates P and J.

Theorem 6.5.5 *BC_sat_MDC*

$$\frac{{}_{\mathsf{BC}.U}\vdash \circlearrowleft J}{J\ {}_{\mathsf{BC}.U}\vdash \mathsf{true} \rightsquigarrow (\forall n, b : n \in A \wedge b \in V : \mathsf{okBC}^{n}_{U,b}.P.(d.b))}$$

Corollary 6.5.6 Self-stabilizing Broadcast

$$\frac{{}_{\mathsf{BC}.U}\vdash \circlearrowleft J}{J\ {}_{\mathsf{BC}.U}\vdash \mathsf{true} \rightsquigarrow (\forall b : b \in V : P.(x.b))}$$

Theorem 6.5.7 *BC_sat_MDCC*

$$\frac{{}_{\mathsf{BC}.U}\vdash \circlearrowleft J}{\begin{array}{ll} J\ {}_{\mathsf{BC}.U}\vdash \mathsf{true} \rightsquigarrow (\forall n, b, b' : & n \in A \wedge b \in V \wedge b' \in N.b : \\ & \mathsf{okBC}^{n}_{U,b}.P.(d.b) \wedge \mathsf{okBC}^{n}_{U,b}.P.(r.b.b')) \end{array}}$$

Figure 6.8: *Main results on the self-stabilizing broadcast program.*

network, provided the condition required by Theorem 6.5.4 is met. In addition, there is a change in that g is no longer a function of type $V \rightarrow A$, but is an array of (input) variables: $g.b$[17] is the input variable of a node b whose value is to be broadcasted. Theorems stating the convergence property of BC are also included in Figure 6.8. Theorem 6.5.5 is a general theorem of self-stabilizing broadcast. Theorem 6.5.7 is a slight variation thereof in which it is also explicitly stated that eventually all communication links will also be stabilized. This latter version will be used later in Chapter 7. Corollary 6.5.6 states it more clearly that all source-data will eventually be broadcasted. The theorems are slightly stronger that the one which directly follows from Theorem 129 6.4.3 in that $g.b$, for $b \in U$, is not required to always satisfy P, but only under the assumption of some stable predicate J. Consequently, only if J holds it is guaranteed the broadcast will perform as expected.

[17] And therefore the environment may control g.

6.6 Some Further Generalization

One condition that keeps appearing is $e_1 = \bot$. Recall that e_1 is the cost of an empty path: $\varrho.b.[] = e_1$. The only reason we insist on the condition is to obtain the δ $\textsc{Self}_{121}$ property:

$$\frac{\mathsf{Lattice}.\sqsubseteq \ \wedge\ (e = \bot)\ \wedge\ b \in U}{\delta.U.b = \bot}$$

Still, assuming $(\forall a, b :: x \sqsubseteq \mathsf{addW}.a.b.x)$ we can conclude that the cost of any path will be at least e_1. Since the empty path is also a path from U to b if $b \in U$, we have:

$$\begin{array}{ll} & \delta.U.b \\ = & \quad \{ \text{ definition}_{119} \text{ of } \delta \ \} \\ & \sqcap\{\varrho.b.s \mid U \xrightarrow{s} b\} \\ = & \quad \{ \ e_1 \sqsubseteq \varrho.b.s \text{ and } U \xrightarrow{[]} b \ \} \\ & e_1 \end{array}$$

Hence:

$$\frac{\mathsf{Lattice}.\sqsubseteq \ \wedge\ b \in U}{\delta.U.b = e_1}$$

If we drop the requirement $e_1 = \bot$, the variable d in the FSA algorithm may initially have a value which is smaller that e_1. Still, the FSA algorithm will self-stabilize. This is nice because then 'negative' initial values may be allowed.

Another re-occurring requirement is $\delta.U.b \neq \top$, which implies that b should be reachable from U. Consider again the FSA algorithm in Figure 5.9_{106}. For the sake of simplicity, let us collect the code and simplify it by dropping the link registers. We obtain the following program:

```
prog    FSA
assign  $([]a, b : a, b \in V : d.a.b := \varphi.\{a\}.b.(d.a))$
```

Let W be the set of nodes in V which are not reachable from a. Let $b \in W$. Evaluating $\varphi.\{a\}.b.(d.a)$ always yields a value greater than $d.a.b'$ of any neighbor b' of b, unless if $b = a$, which is not the case since b is not reachable from a. Consequently, the values of d's in the nodes in W will gradually rise until they hit the $\top$, which is the correct value of $d.a.b$ for all $b \in W$. This is of course assuming the d's will ever hit $\top$, which is only the case if the domain of the lattice $\sqsubseteq$ is finite. So, under this condition, the FSA algorithm will compute the minimum cost from a node a to any other node b, regardless the reachability of b from a.

For the broadcast program the situation is a bit different. Obviously, if b is not reachable from a then no message from a will ever reach b.

Even the night must fail
For light sleeps in the eyes
And dark becomes dark on dark
Until the darkness dies.

Soon the eyes resolves
Complexities of night
Into stillness, where the heart
Falls into fabled light.
— Tas's Song of Courage

Chapter 7 Lifting the FSA Algorithm

LARGE scale computer networks are typically clustered and hierarchically divided: computers are grouped to form domains, and domains are grouped to form larger domains. An example of such a division was displayed in Figure 2.5. As more and more services are offered, and the size of a network increases, so is the bulk of information that has to be maintained to manage the network. At some point —one should imagine a network at a multi-nation scale—, one may decide that some part of the information is more a burden to keep around. For example, one may not want to know the details of the interior of a domain, preferring to keep global information about the domain. With only this kind of partial information, a node may only know, or want to know, how to send messages to a domain. It is then up to the local network in the destination domain to deliver a message to the message's real destination, which is a node, inside the domain. Of course as a trade-off some efficiency may be lost: the local network in the destination domain may turn out to be so slow that the destination node can actually be better approached directly from outside, although this may mean a slight detour. Indeed, people have used hierarchical division for a long time and for various applications, ranging from cataloguing plants to computers networking, and used it not only to organize things, but also to hide information.

A hierarchically network of domains is just an instance of a network of domains in general. The hierarchy is an extra feature which can be exploited to define a *visibility* relation, with the intention that each domain will only have information on those domains visible to it. The others are hidden. The existing network of communication links will be restrained by the visibility relation —hence only communication to visible domains is possible. This induces a new network of domains, which can subsequently be treated without any reference to the hierarchical structure of the domains. Therefore, we will first concentrate on generalizing the FSA algorithm to make it work on networks of domains in general; later it will be discussed how a hierarchically divided network fits in the context.

The only difference between a network of domains and an ordinary network is that a 'node' is now no longer an indivisible unit, but a domain with an interior. There are various ways to distribute domain level information among the nodes that inhabit it, but this is, one can say, more the concern of the communication backbone of the system. When applied at the domain level, we expect the FSA algorithm to be self-stabilizing under the same round solvability condition. If this is indeed the case, the theory developed in Chapter 6 will be re-usable[1]. This approach differs considerably from that of Lentfert [Len93]. Because in the latter no separation is made between those properties which depend on the hierarchical feature and those that do not, the proofs are more complicated.

This chapter will present the aforementioned generalization of the FSA algorithm. Like the FSA algorithm, the generalization has also been verified in HOL. Still, to show the reader some important aspects of the algorithm, and also to provide the reader with another good example of formal program development, a derivation of the algorithm will be presented.

7.1 Domain Level FSA Algorithm

Imagine a network (V, N). The nodes in V represent real computers, and the neighborhood function N represents the physical communication links between the nodes. The nodes can be grouped into domains. Let us say that $\mathcal{V}$ is the set of domains we want to define on top of V and $\mathbf{n}$ is a function, called *interior function*, that expresses the membership of a node with respect to a domain. So, given a domain $\underline{b} \in \mathcal{V}$, $b \in \mathbf{n}.\underline{b}$ means that the node b is inside the domain $\underline{b}$[2]. The neighborhood function N can be lifted to the domain level by defining that a domain $\underline{a}$ is a neighbor of another domain $\underline{b}$ is we have a node a in $\underline{a}$ and a node b in $\underline{b}$ such that those two nodes are linked by N. More precisely, we can define $\mathcal{N} \in \mathcal{V} \to \mathcal{P}(\mathcal{V})$ such that:

$$\underline{a} \in \mathcal{N}.\underline{b} = (\exists a, b : a \in \mathbf{n}.\underline{a} \wedge b \in \mathbf{n}.\underline{b} : a \in N.b)$$

If we define $\mathcal{N}$ as above, then $(\mathcal{V}, \mathcal{N})$ is a new network. It is a network of domains rather than a network of nodes, and $\mathcal{N}$ is the 'projection' of the physical connection at the node level to the domain level.

Sometimes however, we want to have another neighborhood relation at the domain level rather than the $\mathcal{N}$ as defined above. For example, a few paragraphs earlier we have mentioned that in a hierarchically structured network of domains, we may want to restrict $\mathcal{N}$ with a visibility relation. Therefore, in the sequel we will leave $\mathcal{N}$ as *an independent parameter* of a network of domains rather than to have it defined in terms of N.

In the sequel we will describe a network of domains with a tuple $(\mathcal{V}, \mathcal{N}, V, N, \mathbf{n})$ (each parameter is independent). $\mathcal{V}$ is the set of domains in the network, and $\mathcal{N}$ is the neighborhood function at the domain level. $(\mathcal{V}, \mathcal{N})$ is required to form a network —that

[1] In his thesis [Len93] Lentfert distinguished the round solvability condition for the hierarchical networks from the one for ordinary networks. Both conditions are, in our opinion, the same, at least as long as the problem does not rely on the details of the interior of the domains.

[2] Another way to represent a domain is to represent it as a subset of V, rather than using an interior function.

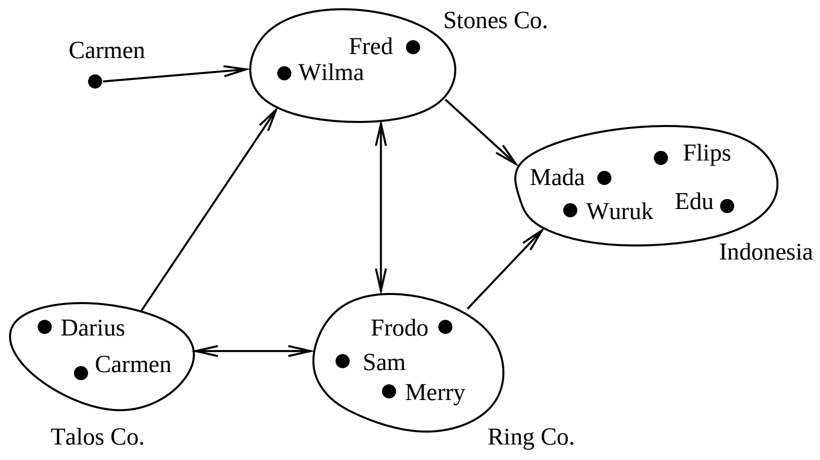


Figure 7.1: *A network of domains.*

is, $\mathcal{N}.\underline{b}$ should be restricted within $\mathcal{V}$, for all $\underline{b} \in \mathcal{V}$. (V, N) is the underlying network of nodes. As said, this reflects the physical communication network. $\mathbf{n} \in \mathcal{V} \to \mathcal{P}(V)$ is the interior function. The domains are typically non-empty and their interiors are finite. They are not required to have disjoint interiors though.

Figure 7.1 displays an example of a network of domains. The reader should bear in mind that the *connectivity function* $\mathcal{N}$ does not have to reflect the physical communication links connecting the domains. Also, $\mathcal{N}$ is not necessarily bi-directional (symmetric). For example, the visibility relation spanned by a hierarchically divided network is typically not symmetric. If $\mathcal{N}$ has to be restricted by the visibility relation, the resulting neighborhood function is also likely to be asymmetric. So, we should be more careful in interpreting the 'direction' induced by $\mathcal{N}$. However, the network spanned by $\mathcal{N}$ is intended to be a subset of the physical network, and therefore whenever two domains are connected by $\mathcal{N}$, we can assume that there are actually physical links —which may even be bi-directional— connecting them.

The reader may notice that we used underlined variables such as $\underline{a}, \underline{b}, \ldots$ to denote domains, whereas nodes were denoted by $a, b, \ldots$. We will keep this convention.

Let in the sequel $(\mathcal{V}, \mathcal{N}, V, N, \mathbf{n})$ *be a network of domains, with interiors.* $\mathcal{V}$ *is finite and non-empty and in addition* $\mathbf{n}.\underline{a}$ *is non-empty for all* $\underline{a} \in \mathcal{V}$. The nodes can be regarded as an abstraction of physical computing units. For each *abstract link* —henceforth it means a link induced by $\mathcal{N}$— we can define its cost. A notion of the minimal cost of going from one domain to another can subsequently be defined. Since a network of domains is just like any other network, except that domains have interiors, any instance, depending on the specific application, of the general minimal cost function δ discussed in Chapter 6 can be used. To distinguish such a δ associated with a network of domains and one associated with an ordinary network we will use 'Δ' to denote the first and reserve 'δ' for the latter. The distinction is of course artificial, but denoting them differently will make things less confusing later. Suppose we are interested in self-stabilizingly computing the value of $\Delta.\underline{\alpha}.\underline{b}$ for all $\underline{b}$ by a given $\underline{\alpha}$[3].

Let in the sequel $\underline{\alpha}$ *be a fixed domain.* We intentionally use a Greek letter α instead of a Latin letter to help the reader later to recognize that $\underline{\alpha}$ can be treated as a constant.

Before we go on discussing how to compute $\Delta.\underline{\alpha}$, there is a question which we wish to address first, namely: having computed $\Delta.\underline{\alpha}.\underline{b}$, where do we keep the result? We can store this centrally in a node in the domain $\underline{b}$ (or $\underline{\alpha}$), or we can let each node in $\underline{b}$ have its own duplicate of $\Delta.\underline{\alpha}.\underline{b}$, or anything between these two extremes. There are good reasons for each choice. Centralized storage is easy to maintain, it saves space, but is prone to failures on the central node. Distributed storage is robust, but less space-efficient and extra effort has to be spent to maintain its consistency. Which one is the best may depend on the application, but in any case the choice does not actually matter for the computation of Δ. Let us assume in the sequel that we have a distributed store. Each node $b \in \mathbf{n}.\underline{b}$ maintains a variable $d.\underline{b}.b$, which eventually should contain the value $\Delta.\underline{\alpha}.\underline{b}$. If we call our program $P.\underline{\alpha}$, its specification will be:

$$\mathsf{true} \ {}_{P.\underline{\alpha}}\!\vdash \mathsf{true} \rightsquigarrow (\forall \underline{b}, b : \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : d.\underline{b}.b = \Delta.\underline{\alpha}.\underline{b}) \tag{7.1.1}$$

Let us now forget about Δ. Let us just say that we want $P.\underline{\alpha}$ to converge to a situation where the values of $d.\underline{b}.b$ for all $\underline{b} \in \mathcal{V}$ and $b \in \mathbf{n}\underline{b}$ satisfy some predicate ok. More specifically, we generalize the specification of $P.\underline{a}$ to:

$$\mathsf{true} \ {}_{P.\underline{\alpha}}\!\vdash \mathsf{true} \rightsquigarrow (\forall n, \underline{b}, b : n \in A \wedge \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)) \tag{7.1.2}$$

As before, A is a domain of rounds. Indeed, we intend that $P.\underline{\alpha}$ converges to its goal above through a fair and successive approximation process along some ordering on the domain of rounds. The domain of the rounds does not have to coincide with the domain of the values of the $d.\underline{b}.b$'s. Let us denote the latter with B. Notice that (7.1.1) can be obtained from (7.1.2) if there exists an $\eta \in A$ such that:

$$\mathsf{ok}^\eta_{\underline{b}}.x \;\Rightarrow\; (x = \Delta.\underline{\alpha}.\underline{b})$$

It is then up to us to find a suitable ok.

In establishing $\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)$ we probably need to evaluate the values of $d.\underline{c}.c$ of other domains $\underline{c}$ and other nodes c. So, values need to be communicated from one domain to another, and from one node to another. Without detailing how they are communicated, let us just say that $\mathsf{cOk}^n_{\underline{b}}$ is the communication obligation of domain $\underline{b}$ at round n. We will strengthen specification (7.1.2) to:

$$\mathsf{true} \rightsquigarrow (\forall n, \underline{b}, b : n \in A \wedge \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : \mathsf{cOk}^n_{\underline{b}} \;\wedge\; \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)) \tag{7.1.3}$$

where —to emphasize what has just been said:

$$\begin{aligned} \mathsf{ok}^n_{\underline{b}}.X &= \textit{the value } X \textit{ is 'acceptable' for domain } \underline{b} \textit{ at round } n \\ \mathsf{cOk}^n_{\underline{b}} &= \textit{the communication obligation of domain } \underline{b} \textit{ at round } n \end{aligned}$$

[3] This is a slightly different problem statement than the one we had in Chapter 5. There, it was required to compute $\Delta.a.b$, for *all* pairs (a, b). However, it has been remarked that $\Delta.a$ can be computed without any information on $\Delta.a'$, for any distinct a'. So, we can as well fix the a. This simplifies things.

In particular, $\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)$ can be viewed as describing the (computation) obligation of (any node in) the domain $\underline{b}$ for the round n.

Let us introduce several abbreviations:

$$\begin{array}{rcl}
\mathsf{dataOk}^n & = & (\forall \underline{b}, b : \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)) \\
\mathsf{comOk}^n & = & (\forall \underline{b} : \underline{b} \in \mathcal{V} : \mathsf{cOk}^n_{\underline{b}}) \\
\mathsf{preOk}^n & = & (\forall m : m \sqsubset n : \mathsf{comOk}^m \wedge \mathsf{dataOk}^m)
\end{array}$$

for some given relation $\sqsubset$ over the domain of the rounds A. *Let in the sequel A be a finite and non-empty set and $\sqsubset$ be a transitive and well-founded relation.*

So, dataOk^n and comOk^n are, respectively, the whole computation and communication obligation for round n, whereas preOk^n can be thought of as describing what has been achieved when a new round n is entered.

Let us not call our program just $P.\alpha$, but $\mathsf{dFSA}.\alpha$, which stands for *Domain-level, Fair, and Successive Approximation.* Using the just introduced abbreviations we can re-state our problem as expressed by (7.1.3) as:

$$\mathsf{DA0}: \quad \mathsf{true} \;{}_{\mathsf{dFSA}.\underline{\alpha}}\!\vdash \mathsf{true} \rightsquigarrow (\forall n : n \in A : \mathsf{comOk}^n \wedge \mathsf{dataOk}^n) \qquad (7.1.4)$$

This is what we are going to consider as *the* specification of the domain-level FSA algorithm.

Just as with the FSA algorithm we can now use ROUND DECOMPOSITION$_{93}$ to reduce the above into a round-based specification, that is, a specification stating where to the system must converge at every round n. *For the sake of readability, confinement constraints will be omitted from formulas.* We derive:

$$\begin{array}{ll}
& \mathsf{true} \vdash \mathsf{true} \rightsquigarrow (\forall n : n \in A : \mathsf{dataOk}^n \wedge \mathsf{comOk}^n) \\
\Leftarrow & \{ (\dagger)\ \text{ROUND DECOMPOSITION}_{93}, \text{Definition of } \mathsf{preOk} \} \\
& (\forall n : n \in A : \mathsf{preOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{dataOk}^n \wedge \mathsf{comOk}^n) \\
\Leftarrow & \{ \text{ACCUMULATION}_{91} \} \\
& (\forall n : n \in A : (\mathsf{preOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{dataOk}^n) \wedge (\mathsf{preOk}^n \vdash \mathsf{dataOk}^n \rightsquigarrow \mathsf{comOk}^n)) \\
\Leftarrow & \{ \rightsquigarrow \text{STABLE SHIFT}_{91} \text{ and STABLE BACKGROUND}_{91} \} \\
& (\forall n : n \in A : (\mathsf{preOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{dataOk}^n) \wedge (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{comOk}^n)) \\
\Leftarrow & \{ \text{definition of } \mathsf{dataOk} \text{ and } \mathsf{comOk}, \rightsquigarrow \text{CONJUNCTION}_{91} \} \\
& (\forall n, \underline{b}, b : \ n \in A \wedge \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : \\
& \qquad (\mathsf{preOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)) \wedge (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \vdash \mathsf{true} \rightsquigarrow \mathsf{cOk}^n_{\underline{b}}))
\end{array}$$

Note that the calculation above is practically the same calculation$_{102}$ as in Chapter 5, the one where we applied ROUND DECOMPOSITION to the specification of the FSA algorithm. This should not be surprising as so far we have not yet exploited aspects which distinguish a network of domains from an ordinary one. Notice that to be able to invoke ROUND DECOMPOSITION in (†) the domain of the rounds A is required to be finite and non-empty, and the relation $\sqsubset$, with which we order the rounds, has to be well-founded. These have been assumed before.

Let us also divide the program $\mathsf{dFSA}.\underline{\alpha}$ into components. The core part of $\mathsf{dFSA}.\underline{\alpha}$ is the computation part. In the FSA algorithm it is the part in which the generators are invoked to re-compute new values of d. To each node b in each domain $\underline{b}$ a computation component $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$ will be associated. The other part of dFSA is the communication part. The role of this part is obvious. To each domain $\underline{b}$ we will associate a communication component $\mathsf{cFSA}.\underline{\alpha}.\underline{b}$[4]. The above calculation results in a round-wise specification, stating that at each round n the program $\mathsf{dFSA}.\underline{\alpha}$ should: (1) converge to $\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)$, and (2) converge to $\mathsf{cOk}^n_{\underline{b}}$. If we insist that the components of dFSA are write-disjoint, then using the TRANSPARENCY$_{92}$ law we can delegate the computation obligation (1) to $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$ and the communication obligation (2) to $\mathsf{cFSA}.\underline{\alpha}.\underline{b}$.

Hence, our original specification DA0 can be refined to DA1 below. For all $n \in A$, $\underline{b} \in \mathcal{V}$ and $b \in \mathbf{n}.\underline{b}$:

$$\begin{array}{ll}
\mathsf{DA1.a:} & \mathsf{dFSA}.\underline{\alpha} \;=\; ([\![\underline{b} : \underline{b} \in \mathcal{V} : \mathsf{cFSA}.\underline{\alpha}.\underline{b}\;]\!]\;([\![b : b \in \mathbf{n}.\underline{b} : \mathsf{gFSA}.\underline{\alpha}.\underline{b}.b) \\
\mathsf{DA1.b:} & P, Q \in \cup\{\{\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b, \mathsf{cFSA}.\underline{\alpha}.\underline{b}\} \mid \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b}\} \;\wedge\; P \neq Q \;\Rightarrow\; P \div Q \\
\mathsf{DA1.c:} & \mathsf{preOk}^n \;{}_{\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b) \\
\mathsf{DA1.d:} & \mathsf{preOk}^n \wedge \mathsf{dataOk}^n \;{}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{cOk}^n_{\underline{b}} \\
\mathsf{DA1.e:} & {}_{\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b}\!\vdash \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n) \\
\mathsf{DA1.f:} & {}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}}\!\vdash \circlearrowleft\mathsf{preOk}^n
\end{array}$$

Notice that some of the above specifications look very similar to $\mathsf{MD3}_{103}$ of the FSA algorithm. Let us first take a look at $\mathsf{DA1.c}$, which is probably the most important part of $\mathsf{DA1}$. The specification requires the value of $d.\underline{b}.b$ to be made 'acceptable' for the domain $\underline{b}$ at the round n. From the calculation of the FSA algorithm in Chapter 5 we learned that, roughly stated, the application of a matching generator Φ to a set of values that are acceptable for all rounds previous to n yields an acceptable value for round n. In other words, it suffices to require round solvability. The following calculation will show this.

First, by exploiting $\rightsquigarrow$ INTRODUCTION$_{91}$ and the definition$_{42}$ of $\mathsf{ensures}$ in much the same way as in the calculation$_{104}$ of $\mathsf{MD3.a}$ we can refine $\mathsf{DA1.c}$ to:

$$\{\mathsf{preOk}^n\}\; a\; \{\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)\} \tag{7.1.5}$$

for *some* $a \in \mathbf{a}(\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b)$ and:

$$ {}_{\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b}\!\vdash \;\; \circlearrowleft\mathsf{preOk}^n \tag{7.1.6}$$

$$ {}_{\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b}\!\vdash \;\; \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)) \tag{7.1.7}$$

Notice that there is nothing in the above specifications that forbids us from making an a satisfying (7.1.5) to be the only action of $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$. So, let us just assume this.

The specification (7.1.7) is now superfluous as it is implied by (7.1.5) and (7.1.6). If cOk is a predicate confined by some set of variables V that is ignored by a, by Corollary 3.4.6$_{33}$ it follows that comOk^n is stable in $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$ for any n:

$$ {}_{\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b}\!\vdash \circlearrowleft\mathsf{comOk}^n \tag{7.1.8}$$

[4] Later, $\mathsf{cFSA}.\underline{\alpha}.\underline{b}$ will be split further to node-level components.

If $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$ only writes to $d.\underline{b}.b$ then $\mathsf{ok}^n_{\underline{c}}.(d.\underline{c}.c)$ is stable, if $(\underline{c}, c)$ and $(\underline{b}, b)$ are distinct. Then, given (7.1.5) one can prove:

$$\{\mathsf{preOk}^n \wedge \mathsf{dataOk}^n\}\ a\ \{\mathsf{dataOk}^n\} \tag{7.1.9}$$

(7.1.8) and the above implies (7.1.6):

$$
\begin{array}{ll}
 & {}_{\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b}\vdash\ \circlearrowright\mathsf{preOk}^n \\
= & \{\ a \text{ is the only action of } \mathsf{gFSA}.\underline{\alpha}.\underline{b}.b, \text{ definition}_{44}\ (4.3.8) \text{ of } \circlearrowright\ \} \\
 & \{\mathsf{preOk}^n\}\ a\ \{\mathsf{preOk}^n\} \\
\Leftarrow & \{\ \text{Hoare triple conjunction, definition}_{143} \text{ of } \mathsf{preOk}\ \} \\
 & (\forall m : m \sqsubset n : (\{\mathsf{preOk}^n\}\ a\ \{\mathsf{comOk}^m\})\ \wedge\ (\{\mathsf{preOk}^n\}\ a\ \{\mathsf{dataOk}^m\})) \\
\Leftarrow & \{\ (\dagger) \text{ definition}_{143} \text{ of } \mathsf{preOk}, \text{ pre-condition strengthening }\} \\
 & (\forall m : m \sqsubset n : (\{\mathsf{comOk}^m\}\ a\ \{\mathsf{comOk}^m\})\ \wedge\ (\{\mathsf{preOk}^m \wedge \mathsf{dataOk}^m\}\ a\ \{\mathsf{dataOk}^m\})) \\
= & \{\ (7.1.8) \text{ and } (7.1.9)\ \} \\
 & \mathsf{true}
\end{array}
$$

So, we conclude that in this case, (7.1.6) too is superfluous. Note by the way that in (†) step in the above calculation $\sqsubset$ needs to be transitive.

This leaves only (7.1.5). We derive (the step marked with †will be motivated later):

$$
\begin{array}{ll}
 & \{\mathsf{preOk}^n\}\ a\ \{\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)\} \\
\Leftarrow & \{\ \text{definition}_{143} \text{ of } \mathsf{preOk} \text{ and } \mathsf{comOk}, \text{ pre-condition strengthening }\} \\
 & \{(\forall m, \underline{b}' : m \sqsubset n \wedge \underline{b}' \in \mathcal{N}.\underline{b} : \mathsf{cOk}^m_{\underline{b}'})\}\ a\ \{\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)\} \\
\Leftarrow & \{\ (\dagger) \text{ introduce a new variable } cp, \text{ insist } \mathsf{cOk}^m_{\underline{b}'} \text{ implies } (\forall \underline{b}, b : \underline{b}' \in \mathcal{N}.\underline{b} \wedge b \in \mathbf{n}.\underline{b} : \\
 & \quad \mathsf{ok}^m_{\underline{b}'}.(\mathsf{snd}.(cp.\underline{b}.b.\underline{b}'))), \text{ pre-condition strengthening }\} \\
 & \{(\forall m, \underline{b}' : m \sqsubset n \wedge \underline{b}' \in \mathcal{N}.\underline{b} : \mathsf{ok}^m_{\underline{b}'}.(\mathsf{snd}.(cp.\underline{b}.b.\underline{b}')))\}\ a\ \{\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)\} \\
\Leftarrow & \{\ \text{choose } a \text{ and introduce } \Phi\ \} \\
 & (a = \mathsf{assign}.(d.\underline{b}.b).(\Phi.\underline{\alpha}.\underline{b}.(\mathsf{snd} \circ (cp.\underline{b}.b)))) \qquad \wedge \\
 & (\forall F :: (\forall m, \underline{b}' : m \sqsubset n \wedge \underline{b}' \in N.\underline{b} :\ \mathsf{ok}^m_{\underline{b}'}.(F.\underline{b}'))\ \Rightarrow\ \mathsf{ok}^n_{\underline{b}}.(\Phi.\underline{\alpha}.\underline{b}.F))
\end{array}
$$

The calculation yields none other than the same round solvability condition$_{105}$ as in the FSA algorithm! The round solvability of minimal-cost functions has been investigated in Chapter 6 and the results can of course be applied here. At this point, we are basically done. There is still the question as to how to establish the round-wise communication obligation. In the FSA algorithm this is done by copying the data in a node to all its neighbors. Now we have domains instead of nodes, so there will be a slight variation to this 'copying' scheme. This will be discussed in the next subsection.

Notice that in the step marked with †in the deviation above, a new variable cp — more precisely, an array of variables— is introduced. The idea is to establish $\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)$ by applying the function $\Phi.\underline{\alpha}.\underline{b}$. Given that the round solvability condition is met, the function still requires values satisfying $\mathsf{ok}^m_{\underline{b}'}$ for all rounds m that proceed n, and neighbors $\underline{b}'$ of $\underline{b}$ to be passed to it. By preOk^n, we have these values kept in $d.\underline{b}'.b'$ for

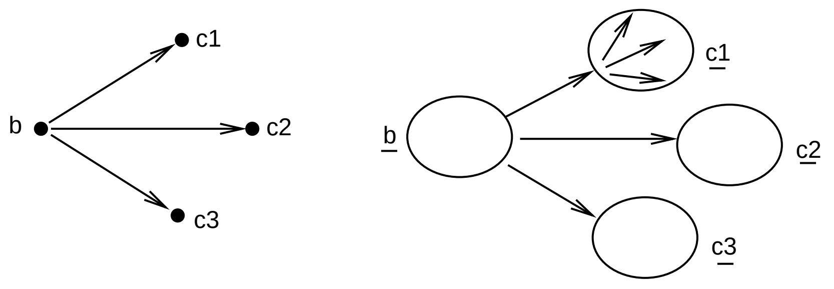


Figure 7.2: *Establishing* cOk.

all neighbors $\underline{b}'$ and nodes $b' \in \mathbf{n}.\underline{b}'$. However, whichever component that maintains $d.\underline{b}'.b'$, it may not be directly linked with the one that keeps $d.\underline{b}.b$. This is why we need to introduce the variable $cp.\underline{b}.\underline{b}'$. It is intended as the 'copy' of $d.\underline{b}'$ and intended to be accessible to the component that maintains $d.\underline{b}'.b'$.

More precisely, the second component of the variable $cp.\underline{b}.\underline{b}'.b$[5] is intended as the view of the node $b \in \mathbf{n}.\underline{b}$ on $d.\underline{b}'.b'$, for some node $b' \in \mathbf{n}.\underline{b}'$. Recall that B is the domain of the values of the variables $d.\underline{b}.b$. *The values of $cp.\underline{b}.\underline{b}'.b$'s are pairs of type $C \times B$, for some type C.* Why they have to be pairs will be explained later. In the FSA algorithm we have the array r for the same purpose, except that in FSA the r's are also the link registers between processes. The cp's are not link registers because now a node b in a domain $\underline{b}$ may not have a direct communication link with a neighboring domain $\underline{b}'$. We will elaborate on this soon. Notice also that in addition, in (†) the predicate cOk has been partially specified.

To summarize the calculation above, we conclude that DA1.c can be refined to DA2 as follows. For all $m, n \in A$, $\underline{a}, \underline{b} \in \mathcal{V}$ and $b \in \mathbf{n}\underline{b}$:

$$\begin{array}{ll}
\text{DA2.a}: & [\mathsf{cOk}^n_{\underline{b}} \;\Rightarrow\; (\forall \underline{c}, c : \underline{c} \in \mathcal{V} \wedge \underline{b} \in \mathcal{N}.\underline{c} \wedge c \in \mathbf{n}.\underline{c} : \mathsf{ok}^n_{\underline{b}}.(\mathsf{snd}.(cp.\underline{c}.c.\underline{b})))] \\
\text{DA2.b}: & \mathsf{cOk}^m_{\underline{a}} \in \mathsf{Pred}.((\mathbf{w}(\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b))^{\mathsf{c}}) \\
\text{DA2.c}: & \mathbf{a}(\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b) \;=\; \{\mathsf{assign}.(d.\underline{b}.b).(\Phi.\underline{\alpha}.\underline{b}.(\mathsf{snd} \circ (cp.\underline{b}.b)))\} \\
\text{DA2.d}: & (\forall F :: (\forall m, \underline{b}' : m \sqsubset n \wedge \underline{b}' \in N.\underline{b} : \; \mathsf{ok}^m_{\underline{b}'}.(F.\underline{b}')) \\
 & \qquad \Rightarrow\; \mathsf{ok}^n_{\underline{b}}.(\Phi.\underline{\alpha}.\underline{b}.F))
\end{array}$$

for some function Φ. Notice that DA2.c practically tells us what the choice of gFSA should be.

7.1.1 Broadcasting Data to Neighboring Domains

Let us now turn our attention to the second most important part of DA1, namely DA1.d:

$$\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \;{}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}}\!\vdash \mathsf{true} \rightsquigarrow \mathsf{cOk}^n_{\underline{b}} \tag{7.1.10}$$

This looks very much like the specification $\mathsf{MD3.b}_{103}$ of the FSA algorithm. In the latter case, the specification requires that at every round n, the value of all the link registers between a node b and its neighbors are made acceptable with respect to the node b. From dataOk^n we know that the datum kept by b is already acceptable, and therefore MD3.b can be met by copying this datum to the link registers. This is illustrated by the picture on the left in Figure 7.2. However, now we have a network of domains instead of an ordinary network. We have decided to use a distributed store, so 'copying data to a domain' will mean, as illustrated by the right picture in Figure 7.2, that we have to broadcast the data across the nodes within the domain.

In the last calculation, a partial specification for cOk is obtained. This is given in DA2.a. In each node $c \in \underline{c}$, a variable $cp.\underline{c}.c.\underline{b}$ is maintained for every neighbor-domain $\underline{b}$ of $\underline{c}$. As said, (the second component of) $cp.\underline{c}.c.\underline{b}$ is intended to be the image of $d.\underline{b}.b$ of some $b \in \mathbf{n}.\underline{b}$. Based on DA2.a, (7.1.10) states that at every round n, all images of $d.\underline{b}$ should be made acceptable with respect to the source-domain $\underline{b}$. Since dataOk^n implies that all $d.\underline{b}.b$'s are already acceptable, this can be achieved by —to re-state what has been hinted before— *broadcasting* the values of $d.\underline{b}.b$'s to all nodes in all neighboring domains. The program BC presented in Chapter 6 is a self-stabilizing broadcast program and can therefore be used for this purpose.

For each target domain, the broadcast will be carried out independently. That is, we are going to divide cOk in (7.1.10) to a number of smaller predicates ccOk's and let a component program converge to each of them independently. More precisely, we insist cOk to have the following form:

$$\mathsf{cOk}^n_{\underline{b}} \;=\; (\forall \underline{c} : \underline{c} \in \mathcal{V} \;\wedge\; \underline{b} \in \mathcal{N}.\underline{c} : \mathsf{ccOk}^n_{\underline{b},\underline{c}})$$

Using $\rightsquigarrow$ Write-disjoint Conjunction$_{92}$ we can refine (7.1.10) or DA1.d to DA3 as follows. For all $\underline{b} \in \mathcal{V}$:

$$\begin{aligned}
&\mathsf{DA3.a}: && \mathsf{cFSA}.\underline{\alpha}.\underline{b} = ([\![\underline{c} : \underline{c} \in \mathcal{V} \;\wedge\; \underline{b} \in \mathcal{N}.\underline{c} : \mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c})\\
&\mathsf{DA3.b}: && (\forall \underline{c}_1, \underline{c}_2 : \; \underline{c}_1, \underline{c}_2 \in \mathcal{V} \;\wedge\; \underline{b} \in \mathcal{N}.\underline{c}_1 \;\wedge\; \underline{b} \in \mathcal{N}.\underline{c}_2 \;\wedge\; (\underline{c}_1 \neq \underline{c}_2) :\\
& && \qquad \mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}_1 \div \mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}_2)
\end{aligned}$$

and for all $n \in A$ and $\underline{c} \in \mathcal{V}$ such that $\underline{b} \in \mathcal{N}.\underline{c}$:

$$\mathsf{DA3.c}: \;\; \mathsf{preOk}^n \wedge \mathsf{dataOk}^n \;{}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}}\!\vdash \; \mathsf{true} \rightsquigarrow \mathsf{ccOk}^n_{\underline{b},\underline{c}}$$

[5] At this point, the increasing number of parameters will start to confuse the reader. Hopefully, this does not discourage him. The essence of the problem is not more complicated than the case is with ordinary networks. It is the details which now become a heavy load to drag around. Alas, there is not much we can do: we still want to maintain some degree of precision in our arguments.

We intend $\mathsf{DA3.c}$ to be implemented by the broadcast program BC, used to broadcast the values of $d.\underline{b}.b$'s in the domain $\underline{b}$ to all nodes in the domain $\underline{c}$. Main theorems about BC were given in Figure 6.7 in Chapter 6. We can use them, but in order to do that we need a definition of ccOk, so that not only the partial specification $\mathsf{DA2.a}$ of cOk is met, but $\mathsf{DA3.c}$ should have the form that matches the conclusion of one of those theorems. Later we will show some calculation to obtain ccOk, but first let us remind the reader of some basic facts about the program BC.

Recall that the program BC silently builds a spanning-tree from the source-nodes to the rest of the network. Data is then broadcasted along the tree. To build the tree, a minimum cost function δ[6] is required.

A communication network is also required. Recall that we have assumed a network of domains $(\mathcal{V}, \mathcal{N}, V, N, \mathbf{n})$. The pair (V, N) represent the physical communication network at the node level, which will be used to do the broadcast. Within each domain, N defines a local network. Recall that we want to broadcast values from the domain $\underline{b}$ to the nodes in $\underline{c}$. To do this we will use the local network of $\underline{c}$. Let us introduce the following abbreviation:

$$(N \restriction \underline{b}).b \;=\; N.b \cap \mathbf{n}.\underline{b} \tag{7.1.11}$$

That is, $(N \restriction \underline{b}).b$ contains all neighbor-nodes of the node b which are inside the domain $\underline{b}$. Note that $(\mathbf{n}.\underline{b}, (N \restriction \underline{b}))$ defines a local network within the domain $\underline{b}$.

We also need source-nodes to broadcast from. What are they in the case of broadcasting data from a domain $\underline{b}$ to a domain $\underline{c}$? A natural choice seems to be $\mathbf{n}.\underline{b}$. However, above we have agreed that $(\mathbf{n}.\underline{c}, N \restriction \underline{c})$ is the network across which data will be spread. The program BC requires that the source-nodes are part of this network. Therefore we will take the so-called *border-nodes* as the source-nodes. That is, those nodes in $\underline{c}$ which have links to nodes in $\underline{b}$. See Figure 7.3. The set of these nodes are denoted by $\mathsf{border}.\underline{c}.\underline{b}$:

$$\mathsf{border}.\underline{c}.\underline{b} \;=\; \{c \mid c \in \mathbf{n}.\underline{c} \wedge (\exists b : b \in \mathbf{n}.\underline{b} \wedge b \in N.c)\} \tag{7.1.12}$$

Now let us return to $\mathsf{DA2.a}$ and $\mathsf{DA3.c}$. To meet $\mathsf{DA2.a}$, it suffices if $\mathsf{ccOk}^n_{\underline{b},\underline{c}}$ implies $(\forall c : c \in \mathbf{n}.\underline{c} : \mathsf{ok}^n_{\underline{b}}.(\mathsf{snd}.(cp.\underline{c}.c.\underline{b})))$. Let us now see what ccOk should be. We should also bear in mind that we should try to come up with a definition that matches with the main theorems about BC given in Figure 6.7. Let $\underline{b} \in \mathcal{V}$, $n \in A$, and let $U = \mathsf{border}.\underline{c}.\underline{b}$. To keep formulas short we will use the following abbreviations:

$$P \;=\; \mathsf{ok}^n_{\underline{b}} \quad\text{and}\quad J \;=\; \mathsf{preOk}^n \wedge \mathsf{dataOk}^n$$

We derive now:

$$(\forall c : c \in \mathbf{n}.\underline{c} : \mathsf{ok}^n_{\underline{b}}.(\mathsf{snd}.(cp.\underline{c}.c.\underline{b})))$$

[6] This is not to be confused with the function Δ mentioned early in this section. Δ is a generic symbol we use to denote a minimum cost function on a network of domains. Computing such a Δ can be the goal of the program dFSA. On the other hand, δ here denotes a minimum cost function on the underlying network of nodes. It will be used by dFSA to broadcast data from one domain to another.

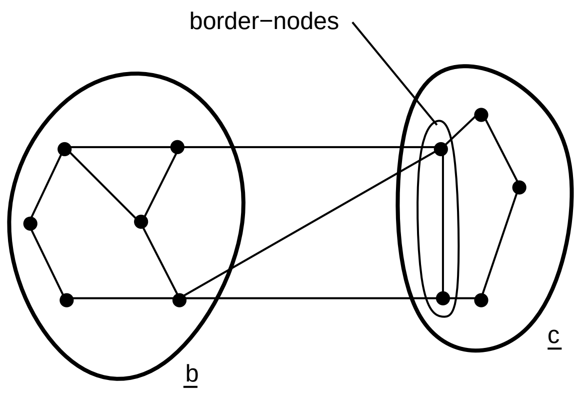


Figure 7.3: *Border-nodes between domains.*

$=\quad \{$ definition of P $\}$

$(\forall c : c \in \mathbf{n}.\underline{c} : P.(\mathsf{snd}.(cp.\underline{c}.c.\underline{b})))$

$=\quad \{$ (†) introduce a lattice $\sqsubseteq_1$, insist $(\forall c : c \in \mathbf{n}.\underline{c} : \delta.U.c \sqsubset_1 \top_1)$ $\}$

$(\forall c : c \in \mathbf{n}.\underline{c} : \delta.U.b \sqsubset_1 \top_1 \ \Rightarrow\ P.(\mathsf{snd}.(cp.\underline{c}.c.\underline{b})))$

$\Leftarrow\quad \{$ definition$_{134}$ of okBC $\}$

$(\forall c : c \in \mathbf{n}.\underline{c} : \mathsf{okBC}^{\top_1}_{c}.P.(cp.\underline{c}.c.\underline{b}))$

So, we can choose the above as $\mathsf{ccOk}^{n}_{\underline{b},\underline{c}}$. If we do so DA3.c becomes:

$$J\ {}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}}\vdash \mathsf{true} \leadsto (\forall c : c \in \mathbf{n}.\underline{c} : \mathsf{okBC}^{\top_1}_{c}.P.(cp.\underline{c}.c.\underline{b})) \qquad (7.1.13)$$

Notice that in (†) it is required that $\delta.U.c \sqsubset_1 \top_1$ for all $c \in \mathbf{n}.\underline{c}$. Since in a lattice everything is less that $\top$, this is equivalent with saying that $\delta.U.c \neq \top_1$, implying that all nodes c in the domain $\underline{c}$ *must be reachable* from the source-nodes in U[7].

The specification (7.1.13) above looks almost like the conclusions of Theorem$_{136}$ 6.5.5 —one of the main theorems of the program BC. However, this is too naive. The program BC uses link-registers r to pass messages from one node to another. So far, nothing is said about the behavior of this r. It is possible that certain initial values of r may destroy the whole convergence process[8]. We have to prove that this is not possible. So, we can expect ccOk to say something about r. The conclusion of another theorem, namely Theorem$_{136}$ 6.5.7, does say something about r:

$$\begin{aligned} J\ {}_{\mathsf{BC}.U}\vdash \mathsf{true} \leadsto (\forall n, b, b' :\ & n \in A \wedge b \in V \wedge b' \in N.b : \\ & \mathsf{okBC}^{n}_{b}.P.(x.b) \wedge \mathsf{okBC}^{n}_{b'}.P.(r.b.b')) \end{aligned} \qquad (7.1.14)$$

[7] Note that only reachability from source-nodes is required. This is weaker than requiring the network of the nodes within a domain to be fully connected.

[8] The reader may wonder why we avoid Corollary$_{136}$ 6.5.6 which seems to assert the self-stabilizing broadcasting ability of BC more explicitly. The reason is that the result, while it is fine, is too sharp for this particular purpose, just like Theorem$_{136}$ 6.5.5 is.

Inspired by the theorem we introduce a new array of variables r, and strengthen the definition of ccOk to:

$$\begin{array}{ll}\mathsf{ccOk}^n_{\underline{b},\underline{c}} & = \\ & (\forall c, c' : \;\; c \in \mathbf{n}.\underline{c} \wedge c' \in (N \restriction \underline{c}).c : \\ & \qquad \mathsf{okBC}^{\top_1}_c.(\mathsf{ok}^n_{\underline{c}}).(cp.\underline{c}.c.\underline{b}) \;\wedge\; \mathsf{okBC}^{\top_1}_{c'}.(\mathsf{ok}^n_{\underline{c}}).(r.\underline{c}.\underline{b}.c.c'))\end{array}$$

Recall that we have introduced the lattice $\sqsubseteq_1$. Let C be the domain of this lattice. We derive:

$$\begin{array}{ll} & \mathsf{ccOk}^n_{\underline{b},\underline{c}} \\ = & \{ \text{ definition } \mathsf{ccOk} \text{ and } P \} \\ & (\forall c, c' : c \in \mathbf{n}.\underline{c} \wedge c' \in (N \restriction \underline{c}).c : \mathsf{okBC}^{\top_1}_c.P.(cp.\underline{c}.c.\underline{b}) \;\wedge\; \mathsf{okBC}^{\top_1}_{c'}.P.(r.\underline{c}.\underline{b}.c.c')) \\ \Leftarrow & \{ \text{ insist } C \text{ to be non-empty } \} \\ & (\forall k, c, c' : k \in C \wedge c \in \mathbf{n}.\underline{c} \wedge c' \in (N \restriction \underline{c}).c : \mathsf{okBC}^{k}_c.P.(cp.\underline{c}.c.\underline{b}) \;\wedge\; \mathsf{okBC}^{k}_{c'}.P.(r.\underline{c}.\underline{b}.c.c'))\end{array}$$

Hence, by $\rightsquigarrow$ SUBSTITUTION$_{91}$ DA3.c can be refined to:

$$\begin{array}{lll} J & {}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}}\vdash & \\ & \mathsf{true} \rightsquigarrow (\forall k, c, c' : & k \in C \wedge c \in \mathbf{n}.\underline{c} \wedge c' \in (N \restriction \underline{c}).c : \\ & & \mathsf{okBC}^{k}_c.P.(cp.\underline{c}.c.\underline{b}) \;\wedge\; \mathsf{okBC}^{k}_{c'}.P.(r.\underline{c}.\underline{b}.c.c'))\end{array} \qquad (7.1.15)$$

The above looks very much like (7.1.14)! In fact, it can be obtained from the latter. Hence, by Theorem$_{136}$ 6.5.7 the above is satisfied by implementing $\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}$ as $\mathsf{BC}.U$.

There is one more remark we wish to add. The program BC requires a function g. For each source-node c, $g.c$ is supposed to tell us where the to-be-broadcasted data of c is stored. In our case here, the source-nodes are the border-nodes between $\underline{c}$ and $\underline{b}$. The source data are the $d.\underline{b}.b$'s stored in the domain $\underline{b}$. Any one of these will do. Since a border-node $c \in \mathsf{border}.\underline{c}.\underline{b}$ will have a neighbor-node b in $\underline{b}$ we can use the corresponding $d.\underline{b}.b$ as $g.c$. Possibly, there are more of such neighbor-nodes, so we will have to pick one. Let us assume a function sel to do the selection:

$$(\exists b : b \in \mathbf{n}.\underline{b} : b \in N.c) \;\Rightarrow\; \mathsf{sel}.c.\underline{b} \in \mathbf{n}.\underline{b} \wedge \mathsf{sel}.c.\underline{b} \in N.c \qquad (7.1.16)$$

The result of applying Theorem$_{136}$ 6.5.7 is summarized in Figure 7.4.

Note that C is the domain of the $\delta.U.c$'s and B is the domain of the values of $d.\underline{b}.b$. When the variable cp is introduced, we insist that $cp.\underline{c}.\_\underline{b}$ has the type of $C \times B$. The reason can be found in the code in Figure 7.4. To broadcast data, we need to compute δ, and this is maintained in the first component of cp.

Let us now discuss the conditions required by Theorem 7.1.1 one by one, and see if they are reasonable.

We need two $\sqcap$-closed lattices and a distributive function addW which also satisfies $k \sqsubset_1 \mathsf{addW}.a.b.k$. These are not too difficult to find. For example, as $\sqsubseteq_1$ we can take the ordering $\leq$ over the interval $[0 \ldots n_{\mathsf{max}}]$, where n_{max} is the number of nodes in the domain $\underline{c}$. For addW we can take the $+1$ function used by the simple notion of minimal

Theorem 7.1.1

Let $\sqsubseteq_1$ and $\sqsubseteq_2$ be two $\sqcap$-closed lattices over, respectively, C and B. Let $\sqcap_3$ belongs to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$. Let $\underline{c} \in \mathcal{V}$ and $\underline{b} \in \mathcal{N}.\underline{c}$. If $\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}$ is defined as:

$$\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c} \;=\; ([\!]c : c \in \mathbf{n}.\underline{c} : \mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}.c)$$

where $\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}.c$ is defined as:

$$\begin{array}{ll}
\mathsf{prog} & \mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}.c \\
\mathsf{read} & \{r.\underline{c}.\underline{b}.c.c' \mid c' \in \mathbf{n}.\underline{c}\} \cup \{r.\underline{c}.\underline{b}.c''.c \mid c'' \in \mathbf{n}.\underline{c}\} \cup \\
 & \{d.\underline{b}.(\mathsf{sel}.c.\underline{b})\} \cup \{cp.\underline{c}.c.\underline{b}\} \\
\mathsf{write} & \{r.\underline{c}.\underline{b}.c''.c \mid c'' \in \mathbf{n}.\underline{c}\} \cup \{cp.\underline{c}.c.\underline{b}\} \\
\mathsf{init} & \mathsf{true} \\
\mathsf{assign} & \\
\multicolumn{2}{l}{cp.\underline{c}.c.\underline{b} := \left\{ \begin{array}{ll} (e_1, d.\underline{b}.(\mathsf{sel}.c.\underline{b})) & \text{, if } c \in \mathsf{border}.\underline{c}.\underline{b} \\ \sqcap_3\{\rtimes\!\ltimes.\mathsf{addW}.c'.c.(r.\underline{c}.\underline{b}.c.c') \mid c' \in (N \restriction \underline{c}).c\} & \text{, otherwise} \end{array} \right.} \\
[\!] & ([\!]c'' : c'' \in \mathbf{n}.\underline{c} \wedge c \in (N \restriction \underline{c}).c'' : r.\underline{c}.\underline{b}.c''.c := cp.\underline{c}.c.\underline{b})
\end{array}$$

for some function $\mathsf{addW} \in V \to V \to C \to C$, then the following holds. For all $n \in A$:

$$\frac{\begin{array}{c}
(\forall a, b :: \mathsf{addW}.a.b \text{ is } \sqcap_1\text{-distributive}) \\
(\forall a, b, k : k \neq \top_1 : k \sqsubset_1 \mathsf{addW}.a.b.k) \\
(\forall c : c \in \mathbf{n}.\underline{c} : \delta.(\mathsf{border}.\underline{c}.\underline{b}).c \neq \top_1) \\
[(\forall c : c \in \mathsf{border}.\underline{c}.\underline{b} : \mathsf{preOk}^n \wedge \mathsf{dataOk}^n \;\Rightarrow\; \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.(\mathsf{sel}.c.\underline{b})))] \\
({}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}}\!\vdash \circlearrowright \mathsf{preOk}^n \wedge \mathsf{dataOk}^n) \;\wedge\; (e_1 = \bot_1) \;\wedge\; (\mathsf{border}.\underline{c}.\underline{b}. \neq \emptyset)
\end{array}}{\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \;\; {}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}}\!\vdash \;\mathsf{true} \rightsquigarrow \mathsf{ccOk}^n_{\underline{b},\underline{c}}}$$

◀

Figure 7.4: *The broadcasting part of* dFSA.

cost. That is: $\mathsf{addW}.a.b.k = k + 1$[9]. As $\sqsubseteq_2$ we can take any $\sqcap$-closed lattice over the domain B of the messages which are to be broadcasted.

The condition $(\forall c : c \in \mathbf{n}.\underline{c} : \delta.(\mathsf{border}.\underline{c}.\underline{b}).c \neq \top_1)$ —as said before— can be interpreted as a condition requiring that all nodes in the domain $\underline{c}$ should be reachable from its' border-nodes. Notice that the network $(\mathbf{n}.\underline{c}, N \restriction \underline{c})$ does not have to be fully connected to satisfy this.

For $\underline{b} \in \mathcal{N}.\underline{c}$, the condition $\mathsf{border}.\underline{c}.\underline{b}. \neq \emptyset$ means that if there are (abstract) links connecting the domain $\underline{b}$ to the domain $\underline{c}$, then there should also be (physical) links connecting some nodes in $\underline{b}$ to some nodes in $\underline{c}$. That is, we require:

$$\underline{b} \in \mathcal{N}.\underline{c} \Rightarrow (\exists b, c : b \in \mathbf{n}.\underline{b} \;\wedge\; c \in \mathbf{n}.\underline{c} : b \in N.c) \tag{7.1.17}$$

This is a reasonable condition.

[9] However, to make it distributive, an exception must be made: $\mathsf{addW}.a.b.n_{\mathsf{max}} = n_{\mathsf{max}}$

There are only two conditions left, namely:

i. $[\ (\forall c : c \in \mathsf{border}.\underline{c}.\underline{b} : \mathsf{preOk}^n \wedge \mathsf{dataOk}^n \Rightarrow \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.(\mathsf{sel}.c.\underline{b})))\ ]$

ii. ${}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}}\vdash \circlearrowright(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n)$

The first is easy. dataOk^n means that for all $\underline{b}$ and $b \in \mathbf{n}.\underline{b}$, $\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)$ holds. Since if $c \in \mathsf{border}.\underline{c}.\underline{b}$ then $\mathsf{sel}.c.\underline{b}$ is a node in $\underline{b}$, we conclude that $\mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.(\mathsf{sel}.c.\underline{b}))$ holds.

For the second, we note that since the component program $\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}$ does not write to the variable d, it cannot destroy dataOk. Hence, it suffices to show the stability of preOk:

$$\mathsf{DA4}: \quad {}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}}\vdash \circlearrowright \mathsf{preOk}^n$$

Notice also that by $\circlearrowright$ COMPOSITIONALITY$_{60}$ the above implies ${}_{\mathsf{cFSA}.\underline{\alpha}.\underline{b}}\vdash \circlearrowright \mathsf{preOk}^n$, which is DA1.f. So, that is one more specification less.

We have now a quite good idea of how the program dFSA is constructed. Its division into gFSA and cFSA has been discussed. The choice of cFSA has been argued above, and the specification DA1 gives a clear indication what gFSA should be. The complete code of dFSA is presented in Figure 7.5. Notice how the component cFSA is broken to node-level components. A compacter code is displayed in Figure 7.6. The code is obtained after collecting the code of the components. It is considerably compacter than Lentfert's original dFSA algorithm in [Len93]. We have removed some redundant parts from the algorithm. Besides, we employ a more concise notation.

7.1.2 Verifying the Specifications (an Overview)

Having given the program in Figure 7.5, there are still some specifications left to argue about. An overview of the specifications we obtain, starting with DA0, and the refinement relation among them is given in Figure 7.7. The specifications which are leaves of the tree are not yet investigated. Let us now go over these one by one.

All specifications regarding the form of the program $\mathsf{dFSA}.\underline{\alpha}$ and the write-disjoint condition of its components are satisfied. This can be directly verified. These specifications are DA1.a, DA1.b, DA2.c, DA3.a, and DA3.b. Specification DA2.a is a partial definition for cOk, and is met by the choice of cOk and ccOk in page 147 and 150. Specification DA2.d is the round solvability condition which the generator Φ is required to meet. Whether or not this is the case depends on the nature of the original problem the program dFSA is required to solve, not on the code of dFSA itself. DA2.b is met because the predicate cOk only refers to the variable cp and r whereas the program gFSA only writes to d.

DA1.e requires the stability of $\mathsf{preOk}^n \wedge \mathsf{dataOk}^n$ in the component $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$. By unfolding the definition$_{143}$ of dataOk, $\mathsf{preOk}^n \wedge \mathsf{dataOk}^n$ is equal to:

$$\mathsf{preOk}^n \wedge \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b) \wedge (\forall \underline{b}', b' : \underline{b}' \in \mathcal{V} \wedge b' \in \mathbf{n}.\underline{b}' \wedge (\underline{b}', b' \neq \underline{b}, b) : \mathsf{ok}^n_{\underline{b}'}.(d.\underline{b}'.b'))$$

The stability of $\mathsf{preOk}^n \wedge \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b)$ in $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$ follows from (7.1.7) whose correctness has been argued. The stability of the rest follows from the fact that $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$ only writes to $d.\underline{b}.b$.

The program $\mathsf{dFSA}.\underline{\alpha}$ is defined as:

$$\mathsf{dFSA}.\underline{\alpha} \;=\; (\Box\underline{b}, b : \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : \mathsf{gFSA}.\underline{\alpha}.\underline{b}.b) \;\Box\; (\Box\underline{b} : \underline{b} \in \mathcal{V} : \mathsf{cFSA}.\underline{\alpha}.\underline{b})$$

where $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$ is defined as:

```
prog    $\mathsf{gFSA}.\underline{\alpha}.\underline{b}.b$
read    $\{cp.\underline{b}.b.\underline{b}' \mid \underline{b}' \in \mathcal{V}\} \cup \{d.\underline{b}.b\}$
write   $\{d.\underline{b}.b\}$
init    true
assign  $d.\underline{b}.b := \Phi.\underline{\alpha}.\underline{b}.(\mathsf{snd} \circ (cp.\underline{b}.b))$
```

and where:

$$\mathsf{cFSA}.\underline{\alpha}.\underline{b} \;=\; (\Box\underline{c} : \underline{b} \in \mathcal{N}.\underline{c} : \mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c})$$
$$\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c} \;=\; (\Box c : c \in \mathbf{n}.\underline{c} : \mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}.c)$$

where $\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}.c$ is defined as:

```
prog    $\mathsf{cFSA}.\underline{\alpha}.\underline{b}.\underline{c}.c$
read    $\{r.\underline{c}.\underline{b}.c.c' \mid c' \in \mathbf{n}.\underline{c}\} \cup \{r.\underline{c}.\underline{b}.c''.c \mid c'' \in \mathbf{n}.\underline{c}\} \cup$
        $\{d.\underline{b}.(\mathsf{sel}.c.\underline{b})\} \cup \{cp.\underline{c}.c.\underline{b}\}$
write   $\{r.\underline{c}.\underline{b}.c''.c \mid c'' \in \mathbf{n}.\underline{c}\} \cup \{cp.\underline{c}.c.\underline{b}\}$
init    true
assign
$cp.\underline{c}.c.\underline{b} := \begin{cases} (e_1, d.\underline{b}.(\mathsf{sel}.c.\underline{b})) & \text{, if } c \in \mathsf{border}.\underline{c}.\underline{b} \\ \sqcap_3\{>\!\!\!-\!\!\!<.\mathsf{addW}.c'.c.(r.\underline{c}.\underline{b}.c.c') \mid c' \in (N \upharpoonright \underline{c}).c\} & \text{, otherwise} \end{cases}$
$\Box$      $(\Box c'' : c'' \in \mathbf{n}.\underline{c} \wedge c \in (N \upharpoonright \underline{c}).c'' : r.\underline{c}.\underline{b}.c''.c := cp.\underline{c}.c.\underline{b})$
```

where $\sqcap_3$ belongs to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$, for some lattices $\sqsubseteq_1$ and $\sqsubseteq_2$.

Figure 7.5: *The domain-level* FSA *algorithm.*

```
prog    $\mathsf{dFSA}.\underline{\alpha}$
init    true
assign
        $(\Box\underline{b}, b : \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : d.\underline{b}.b := \Phi.\underline{\alpha}.\underline{b}.(\mathsf{snd} \circ (cp.\underline{b}.b)))$
$\Box$      $(\Box\ \underline{b}, \underline{c}, c : \underline{c} \in \mathcal{V} \wedge \underline{b} \in \mathcal{N}.\underline{c} \wedge c \in \mathbf{n}.\underline{c} :$
                  if $c \in \mathsf{border}.\underline{c}.\underline{b}$ then $cp.\underline{c}.c.\underline{b} := (e_1, d.\underline{b}.(\mathsf{sel}.c.\underline{b}))$
                  else $cp.\underline{c}.c.\underline{b} := \sqcap_3\{>\!\!\!-\!\!\!<.\mathsf{addW}.c'.c.(r.\underline{c}.\underline{b}.c.c') \mid c' \in N.c\}$
            $\Box$  $(\Box c'' :) c'' \in \mathbf{n}.\underline{c} \wedge c \in N.c'' : r.\underline{c}.\underline{b}.c''.c := cp.\underline{c}.c.\underline{b})$
```

Figure 7.6: *The domain-level* FSA *algorithm —the code has been collected.*

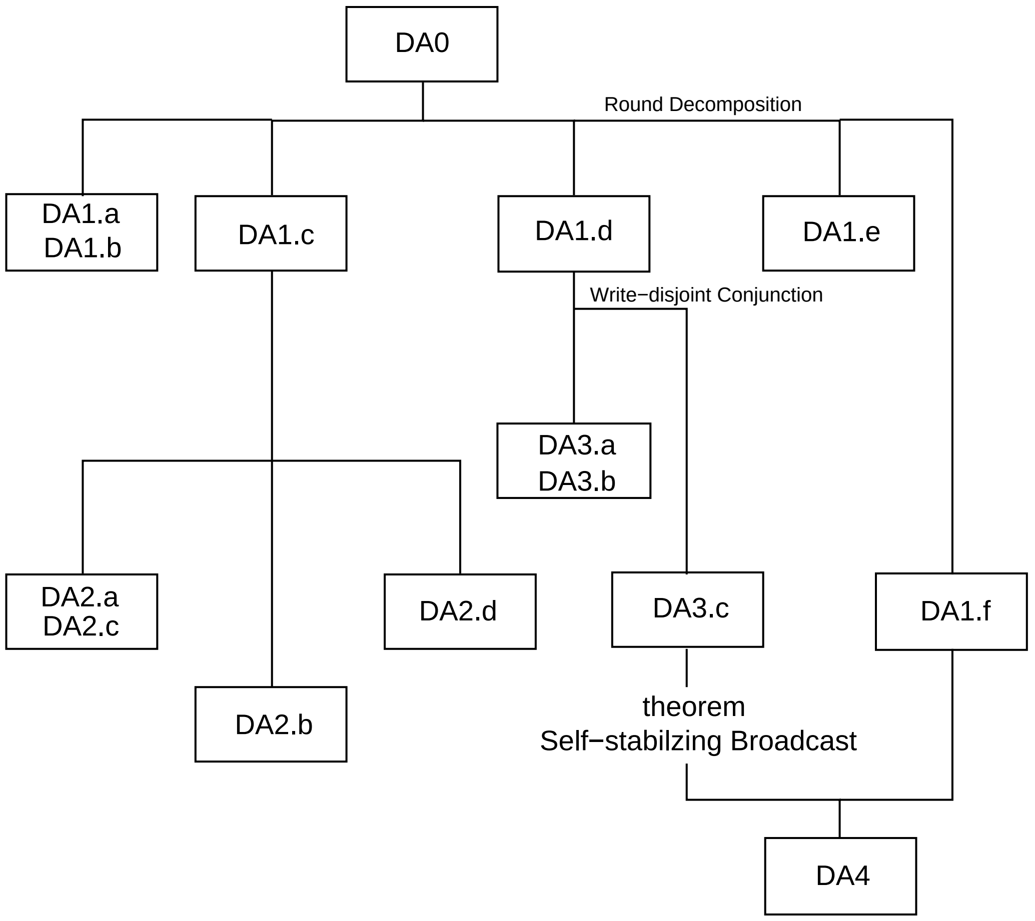


Figure 7.7: *The refinement scheme of* DA0.

So, this leaves only DA4. We will stop at this point. Presenting the verification of DA4 will not reveal anything special which has not been mentioned so far —and it is also awfully tedious. The reader is assured that we have mechanically verified the final result, which will be presented in the next subsection.

7.1.3 The Final Result

As a final result, we conclude that the program dFSA, under all conditions mentioned in the previous subsections, satisfies the specification$_{143}$ DA0, and hence also (7.1.2)$_{142}$. A summary of all those conditions is presented in the theorem in Figure 7.8. Notice that the last condition, labelled by RS, asserts the round solvability of the problem. Notice that it is the same round solvability condition as required by the FSA algorithm for ordinary networks.

If the problem is to compute $\Delta.\underline{b}$, for all domains $\underline{b}$ and for a given function $\Delta \in$

Theorem 7.1.2

Let the network of domains $(\mathcal{V}, \mathcal{N}, V, N, \mathbf{n})$, the $\sqcap$-closed lattices $\sqsubseteq_1 \in C \to C \to \mathbb{B}$ and $\sqsubseteq_2 \in B \to B \to \mathbb{B}$, the constant e_1, and function $\mathsf{addW} \in V \to V \to B \to B$ be the ones used in the program dFSA in Figure 7.5. Let A be a domain of rounds, ordered by $\sqsubset$. If they satisfy the following conditions:

i. $\mathcal{V}$, V, A and C are non-empty and finite, and so is $\mathbf{n}.\underline{b}$, for all $\underline{b} \in \mathcal{V}$.

ii. $\sqsubset$ is transitive and well-founded.

iii. $(\forall a, b :: \mathsf{addW}.a.b$ is $\sqcap_1$-distributive) and $(\forall a, b, k : k \neq \top_1 : k \sqsubset_1 \mathsf{addW}.a.b.k)$.

iv. $(\forall \underline{b}, \underline{c}, c : \underline{c} \in \mathcal{V} \wedge \underline{b} \in \mathcal{N}.\underline{c} \wedge c \in \mathbf{n}.\underline{c} : \delta.(\mathsf{border}.\underline{c}.\underline{b}).c \neq \top_1)$.

v. $e_1 = \bot_1$ and $\top_1 \in C$

vi. $(\forall \underline{b}, \underline{c} : \underline{b}, \underline{c} \in \mathcal{V} : \underline{b} \in \mathcal{N}.\underline{c} \Rightarrow \mathsf{border}.\underline{c}.\underline{b} \neq \emptyset)$.

and in addition, the generator Φ satisfies:

$$\mathsf{RS}: \quad (\forall F :: (\forall m, \underline{b}' : m \sqsubset n \wedge \underline{b}' \in \mathcal{N}.\underline{b} : \mathsf{ok}^m_{\underline{b}'}.(F.\underline{b}')) \Rightarrow \mathsf{ok}^n_{\underline{b}}.(\Phi.\underline{\alpha}.\underline{b}.F))$$

for all $n \in A$ and $\underline{b} \in \mathcal{V}$, and for *a given* predicate ok, then the program $\mathsf{dFSA}.\underline{\alpha}$ will satisfy:

$$\mathsf{true} \vdash \mathsf{true} \rightsquigarrow (\forall n, \underline{b}, b : n \in A \wedge \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b))$$

Figure 7.8: *The main result of the domain-level FSA algorithm.*

$\mathcal{V} \to B$, then the predicate ok should be chosen in such a way that:

$$(\forall n, \underline{b}, b : n \in A \wedge \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b))$$

implies $(\forall \underline{b}, b : \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : d.\underline{b}.b = \Delta.\underline{b})$. This is met —to repeat what has been said early in this section— if there exists a $\eta \in A$ such that:

$$\mathsf{ok}^\eta_{\underline{b}}.x \Rightarrow (x = \Delta.\underline{b})$$

If $\Delta = \delta.\underline{\alpha}$, for some minimal cost function δ, then we already know something as the round solvability of this kind of function has been investigated in Chapter 6, and the results are of course applicable here.

7.2 Hierarchically Divided Network of Domains

In this section we will discuss the applicability of the domain-level FSA algorithm presented in the previous section on networks of domains which are hierarchically divided. The motivation behind such a division has been discussed before.

A hierarchical division of domains is a tree defined over the domains. If a domain $\underline{b}$ is a descendant of another domain $\underline{c}$ then $\underline{b}$ is said to have a lower hierarchy than $\underline{c}$. The leaves of the tree are nodes, which form the interior of the domains.

A tree is can be described by a triple $(V, \lhd, \Omega)$ where V describes the elements of the tree, Ω is the root, and $\lhd$ describes the 'sons' relationship in the tree. That is, $x \lhd y$ means that x is a son of y. The transitive closure of $\lhd$, denoted by $\lhd+$ describes the *proper descendant* relation among the domains, whereas the reflexive and transitive closure of $\lhd$, that is, $\lhd*$, describes the *descendant* relation among the domains.

Let us say that we have a network of nodes (V, N). This network reflects a physical communication network among the nodes. We are also given a set of domains $\mathcal{V}$ containing V, and a tree $(\mathcal{V}, \lhd, \Omega)$, such that V form the leaves of the tree. The tree defines a hierarchy upon the domains. An interior function $\mathbf{n}$ can be defined:

$$\mathbf{n}.\underline{b} = \{b \mid b \in V \wedge b \lhd\!* \underline{b}\} \tag{7.2.1}$$

The function $\mathbf{n}$ describes which nodes 'inhabit' a domain. Note that $b \in V = (\mathbf{n}.V = \{b\})$. Note also that $\lhd*$ satisfies:

$$\underline{b} \lhd\!* \underline{c} \;=\; \mathbf{n}.\underline{b} \subseteq \mathbf{n}.\underline{c}$$

and that $\underline{N}$ satisfies:

$$\underline{b} \in \underline{N}.\underline{c} \;\Rightarrow\; (\mathsf{border}.\underline{c}.\underline{b} \neq \emptyset) \tag{7.2.2}$$

which is one of the condition required by the domain-level FSA algorithm. Any function $\mathcal{N}$ 'smaller' than $\underline{N}$ —that is, $\mathcal{N}.\underline{b} \subseteq \underline{N}.\underline{b}$, for all $\underline{b} \in \mathcal{V}$— will also satisfy above.

In addition the network N also defines a network at the domain level. We can define:

$$\underline{b} \in \underline{N}.\underline{c} = (\exists b, c : b \in \mathbf{n}.\underline{b} \wedge c \in \mathbf{n}.\underline{c} : b \in N.c) \tag{7.2.3}$$

for all $\underline{b}, \underline{c} \in \mathcal{V}$. The network $(\mathcal{V}, \underline{N})$ fully describes the underlying communication network. The idea is to restrict this network with a relation defining 'visibility' among the domains, that is, defining which domains can be seen from a given domain. If a domain is not visible, then no direct information on it will be available. The idea behind this information hiding has been motivated before. A hierarchical division induces a notion of visibility, although as far as the FSA algorithm is concerned, any notion of visibility will do. The algorithm does not require much of the network on which it is based. Only that the border between two domains should not be empty. However, above it has been remarked that any network smaller than the 'full' network, that is the one spanned by $\underline{N}$, satisfies this requirement. If we are interested in computing a minimal cost function δ, the theorems in Chapter 6 guarantee correct results for all $\delta.\underline{\alpha}.\underline{b}$'s such that $\underline{b}$ is reachable from $\underline{\alpha}$. However, at the end of Chapter 6 it has been remarked that even this reachability condition can be dropped —if the lattice being used is finite.

Having said that, let us now take a look at the notion of visibility —as used by Lentfert in his hierarchical FSA algorithm— induced by a hierarchical division. We will then discuss the precise instantiation of the program $\mathsf{dFSA}.\underline{\alpha}$ to this specific problem.

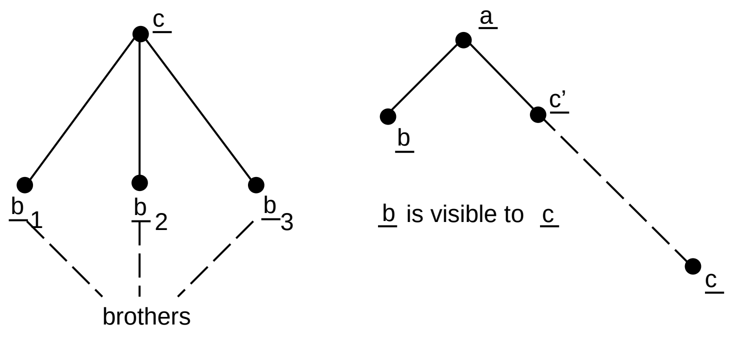


Figure 7.9: *brother and visible domain*

7.2.1 Visibility

Two domains are said to be *brothers*, denoted by $\underline{b} \curlywedge \underline{c}$, if they are unequal and have a common parent. A domain $\underline{b}$ is said to be *visible* from $\underline{c}$, denoted by $\underline{b} \curvearrowleft \underline{c}$ if $\underline{c}$ has an ancestor which is a brother of $\underline{b}$. Consequently, brothers are pair-wise visible, and if a domain is visible to another, the one cannot be a descendant of the other. See the illustration in Figure 7.9. What is the idea behind this choice of visibility notion? It is decided that any domain $\underline{b}$ should have *some* knowledge of what lies under any super-domain —that is, an ancestor domain of $\underline{b}$. This knowledge will not be complete —otherwise $\underline{b}$ will have a complete knowledge of the whole system because the root domain Ω is also a super-domain of $\underline{b}$. It is decided that for each super-domain, we can only go one step down, and that is all that we know about it. So, we know everything about the sons of this super-domain, but not beyond.

Given a tree $(\mathcal{V}, \lhd)$, below is the formal definition of $\curlywedge$ and $\curvearrowleft$.

Definition 7.2.1 Brothers

$$\underline{b} \curlywedge \underline{c} \;=\; (\underline{b} \neq \underline{c}) \wedge (\exists \underline{a} :: \underline{b} \lhd \underline{a} \wedge \underline{c} \lhd \underline{a})$$

Definition 7.2.2 Visibility

$$\underline{b} \curvearrowleft \underline{c} \;=\; (\exists \underline{c}' : \underline{b} \curlywedge \underline{c}' \wedge \underline{c} \lhd\!* \underline{c}')$$

In Figure 7.10 a tree is displayed. There is one point in the tree marked with $\underline{\alpha}$. The darker area covers those domains to which $\underline{\alpha}$ is visible, and the lighter area covers the domains to which $\underline{\alpha}$ is not visible. Since $\underline{\alpha}$ itself is outside the dark area, we cannot expect the relation $\curvearrowleft$ to be symmetric, and hence restricting the $\underline{N}$ with $\curvearrowleft$ will not induce a relation which is bi-directional. This is the reason that we have, since the beginning, avoided to rely the calculation on the bi-directionality. Admittedly,

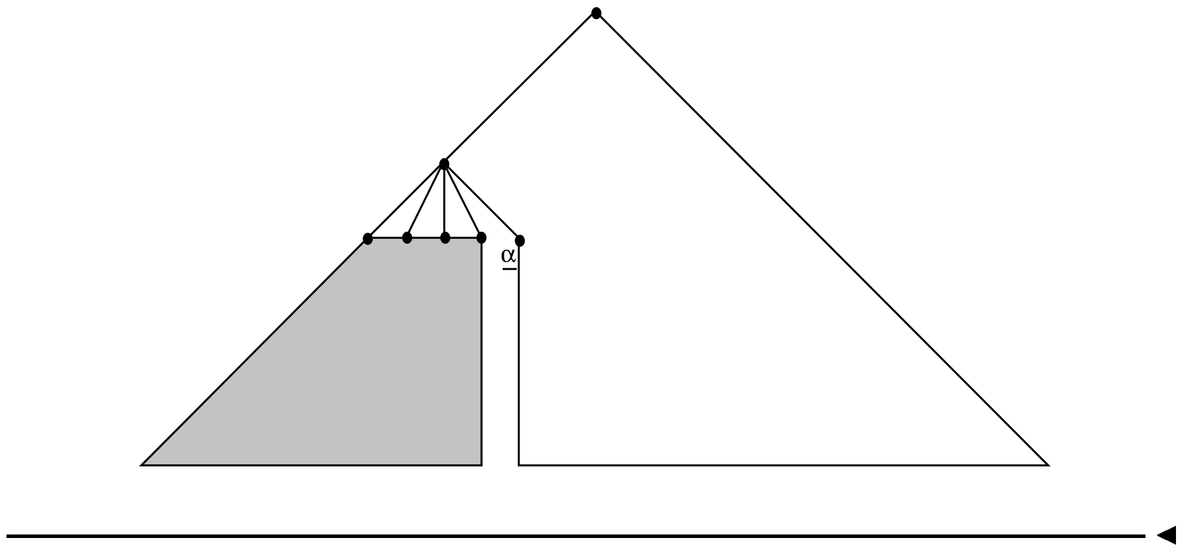


Figure 7.10: *Area of domains that can see $\underline{\alpha}$.*

bi-directionality is useful to simplify arguments, and we did abuse this by sometimes using terms such as 'links between a and b' or 'distance between a and b' as if we have a bi-directional network.

The visibility relation is also not transitive. Look again at the picture to the right in Figure 7.9. We have $\underline{c}' \curvearrowleft \underline{b}$ (because they are brothers) and $\underline{b} \curvearrowleft \underline{c}$, but not $\underline{c}' \curvearrowleft \underline{c}$. A transitive visibility relation —such a relation is often preferred since it has interesting algebraic properties— can be obtained by re-defining the brotherhood relation by allowing a domain to be a brother of itself:

$$\underline{b} \curlywedge \underline{c} \;=\; (\exists \underline{a} :: \underline{b} \lhd \underline{a} \wedge \underline{c} \lhd \underline{a})$$

A consequence of this choice is that all super-domains will become visible.

7.2.2 Hierarchical FSA Algorithm

Let (V, N) be a network reflecting the physical communication network among the nodes. Let $\mathcal{V}$ be a set of domains, containing V. Let $(\mathcal{V}, \lhd, \Omega)$ be a tree defining the hierarchical division on the domains. Let $\mathbf{n}$ and $\underline{N}$ be defined as in (7.2.1) and (7.2.3). As said, the quadruple $(\mathcal{V}, \underline{N}, V, \mathbf{n})$ defines a network of domains. To restrict this network with the visibility relation we define $\mathcal{N}$:

$$\mathcal{N}.\underline{b} \;=\; \{\underline{b}' \mid \underline{b}' \in \underline{N}.\underline{b} \wedge \underline{b}' \curvearrowleft \underline{b}\} \tag{7.2.4}$$

The quadruple $(\mathcal{V}, \mathcal{N}, V, \mathbf{n})$ defines a new network of domains. As an example of how a walk through this network looks like, see Figure 7.11. It shows a hierarchy of domains with the nodes forming the leaves. The broken lines depicts bi-directional communication links between the nodes. Consider the sequence $\underline{e}; \underline{d}; d; c; \underline{a}; \underline{\alpha}$. Each domain in this sequence is visible to its predecessor. So, it is a path from $\underline{\alpha}$ to $\underline{e}$ in the network spanned by $\curvearrowleft$. The path also has a corresponding path at the node level,

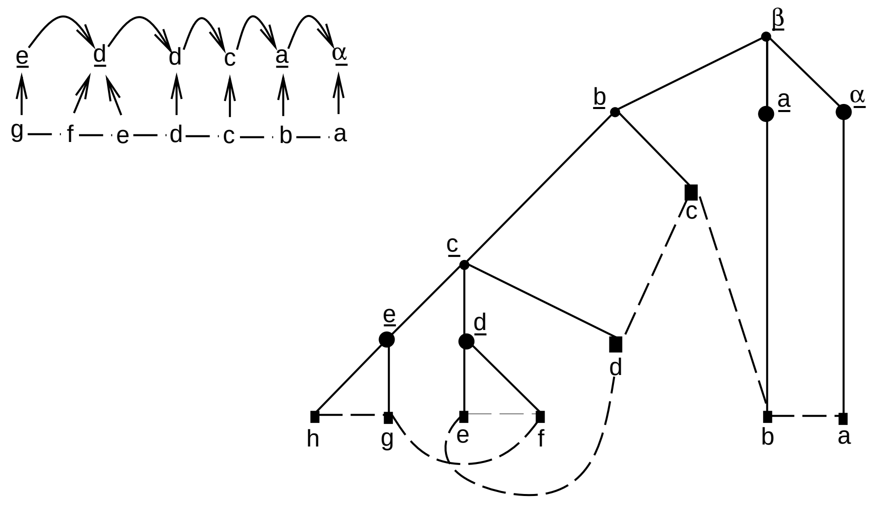


Figure 7.11: *A walk trough a visibility-constrained network.*

for example: $g; f; e; d; c; b; a$. Each node in this sequence is physically linked to its predecessor. How to map this sequence to the sequence $\underline{e}; \underline{d}; d; c; \underline{a}; \underline{\alpha}$ is illustrated by the diagram on the upper left in Figure 7.11. Each node, mapped with an arrow to a domain, is a member of that domain. So, the sequence $\underline{e}; \underline{d}; d; c\underline{a}; \underline{\alpha}$ is a path in a network spanned by $\underline{N}$. So, we conclude that it is also a path in a network spanned by $\mathcal{N}$.

If we now apply the domain-level FSA algorithm on the network $(\mathcal{V}, \mathcal{N}, V, \mathbf{n})$ then we obtain a hierarchical FSA algorithm, and that is all that it takes. We are done, but before we close this chapter, let us add a few comments about how to interpret the program dFSA in this context, and about hierarchical cost functions.

Let $\underline{\alpha}$ be a domain in. The compact version of the program dFSA.$\underline{\alpha}$ is re-displayed below. Some parameters will be renamed so they will not conflict with the names introduced above.

```
prog    dFSA.$\underline{\alpha}$
init    true
assign
        $([]\underline{b}, b : \underline{b} \in \mathcal{V}_{\underline{\alpha}} \wedge b \in \mathbf{n}.\underline{b} : d.\underline{b}.b := \Phi.\underline{\alpha}.\underline{b}.(\mathsf{snd} \circ (cp.\underline{b}.b)))$
[]      $([]\ \underline{b}, \underline{c}, c : \underline{c} \in \mathcal{V}_{\underline{\alpha}} \wedge \underline{b} \in \mathcal{N}_{\underline{\alpha}}.\underline{c} \wedge c \in \mathbf{n}.\underline{c} :$
                if $c \in$ border.$\underline{c}.\underline{b}$ then $cp.\underline{c}.c.\underline{b} := (e_1, d.\underline{b}.(\mathsf{sel}.c.\underline{b}))$
                else $cp.\underline{c}.c.\underline{b} := \sqcap_3\{\bowtie.\mathsf{addW}.c'.c.(r.\underline{c}.\underline{b}.c.c') \mid c' \in N.c\}$
           [] $([]c'' :) c'' \in \mathbf{n}.\underline{c} \wedge c \in N.c'' : r.\underline{c}.\underline{b}.c''.c := cp.\underline{c}.c.\underline{b})$
```

Theorem 7.1.2 states that the program dFSA.$\underline{\alpha}$ satisfies:

$$\text{true} \vdash \text{true} \leadsto (\forall n, \underline{b}, b : n \in A \wedge \underline{b} \in \mathcal{V}_{\underline{\alpha}} \wedge b \in \mathbf{n}.\underline{b} : \mathsf{ok}^n_{\underline{b}}.(d.\underline{b}.b))$$

For each domain $\underline{b} \in \mathcal{V}_{\underline{\alpha}}$ the program can be thought as trying to obtain or to construct some information about the domain $\underline{\alpha}$. For example, it can be that the algo-

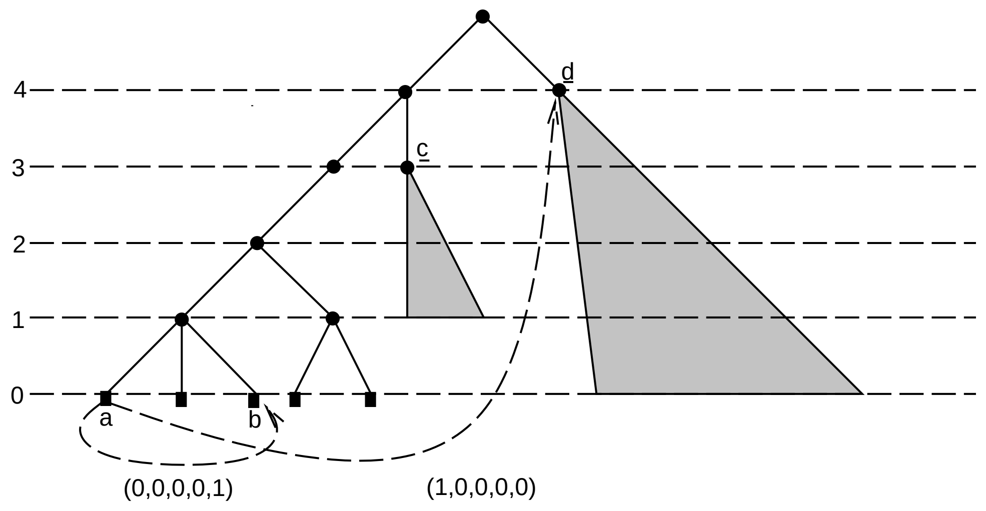


Figure 7.12: *Hierarchical cost.*

rithm is trying to compute $\delta.\underline{\alpha}.\underline{b}$ for each $d.\underline{b}.b$ —δ is some minimal distance function. Because the philosophy of introducing the notion of visibility is to keep only information of visible objects, then $\underline{\alpha}$ has to be visible to $\underline{b}$. So, the set $\mathcal{V}_{\underline{\alpha}}$ is the set of all domains to which $\underline{\alpha}$ is visible —the darker area in Figure 7.10— plus $\underline{\alpha}$ itself. For the connectivity function $\mathcal{N}_{\underline{\alpha}}$ we can use $\mathcal{N}$, restricted to $\mathcal{V}_{\underline{\alpha}}$. Note that even if the underlying network (V, N) at the node level is connected, the network $(\mathcal{V}, \mathcal{N})$ is definitely not: with $\mathcal{N}$, only domains at the same or lower levels can be reached. One should take this into account if he is given a problem that relies heavily on the connectivity of the network being used. In any case, because $\underline{\alpha}$ is visible to all domains in $\mathcal{V}_{\underline{\alpha}}$ (except for $\underline{\alpha}$ itself), then if *within every domain* the nodes are connected, we can expect that all those domains will be reachable from $\underline{\alpha}$. For a large class of problems, broadcasting being an example, no full connectivity is required, and instead, reachability from a given source node is sufficient.

As said, the algorithm can be used to compute the minimal cost of going from one domain to another. For the algorithm itself it does not matter what choice of the minimal cost function is taken, as long as it is round solvable. The choice depends on how one defines the cost of a link, how one wants to 'sum' the cost, and how one defines 'minimum'. At the beginning of Chapter 6 some examples of link-cost were mentioned. It can be, for example, simply a constant 1, or, in a more sophisticated case, a vector carrying various information. In the case of a hierarchically divided network, one may want to define the cost in such a way that the cost of going to a higher level domain is higher, for example because the involvement of a higher level domain may mean the involvement of all nodes inside it, and higher level domains do tend to have more nodes. Lentfert uses a vector [Len93] with the size of the height of the hierarchical tree being used. See Figure 7.12. The cost of a link from a to b is the vector $(0, 0, 0, 0, 1)$ because the target domain b is on level 0. On the other hand, the cost of a link from

a to $\underline{c}$ is $(0,1,0,0,0)$ and to $\underline{d}$ is $(1,0,0,0,0)$ since the target domains are on level, respectively, 3 and 4. To give a greater weight on higher domains, the vectors are lexicographically ordered, from left to right. The 'sum' function used by Lentfert is the standard $+$, point-wisely applied to each component of the vectors. This is a pretty standard operation. The function addW that is obtained can be expected to satisfy the conditions discussed in Chapter 6, required for the round solvability of the resulting minimal cost function.

Part II

Mechanical Verification of Self-Stabilizing Programs

The robots! They're coming this way!

Chapter 8

Mechanical Theorem Proving with HOL

The HOL system will be briefly introduced: it will be explained how to write a formula in HOL, how to add definitions, and how to prove a theorem.

LET us first make an inventory of what we have done so far. In Part I of this thesis we have presented an extension of the programming logic UNITY. The extension has been proven to support stronger compositionality results. We have also formalized the notion of self-stabilization —or more generally: convergence— in UNITY and presented its various basic properties. The theories were applied to several examples, the largest example being Lentfert's FSA algorithm. The algorithm was motivated by the problem of self-stabilizingly computing the minimal distance (cost) between any pair of nodes in a network. The problem seems trivial. Besides, minimal cost is an old problem which has been addressed many times before. However, proving that it can be self-stabilizingly computed is a different problem, and definitely not a trivial one. The FSA algorithm is a general, distributed algorithm that can self-stabilize the underlying network of processes to some common goal, provided the so-called *round solvability* condition is met. The round solvability of minimal cost functions was also thoroughly investigated. Finally, a generalization of the FSA algorithm is presented so that it can be applied to clustered networks —that is, network in which nodes are grouped to form domains— in general, and hierarchically divided networks in particular.

Many of the above mentioned results are useful —and not only for the specific applications addressed in this thesis. However, these are not really the main contribution of our research. Our major contribution is the fact that we have mechanically verified (most) results mentioned above, using the theorem prover HOL. It is however quite impractical to present the complete mechanical proofs we produced in this thesis, and discuss them step by step. So instead, in this second part we would like to take the reader on a short trip into the realm of mechanical verification. In this chapter, we will provide a brief introduction to the HOL system. Those who are familiar with HOL may wish to skip this chapter, except perhaps Subsection 8.3.1 in which a tool that we wrote to support an equational proof style is discussed. This chapter is not going to be a tutorial for the HOL system. We will briefly show how formulas are written in HOL, and how proofs are written and engineered in HOL. A complete introduction to HOL can be found in [GM93]. In some places we also insert our personal opinions about the HOL system. The reader should not be discouraged if the comments are not all positive. HOL is a very potential system. It is, one can say, one of the best available general purpose theorem prover at the moment. Still, a lot of work has yet to be done to improve the system —the user interface, automatic decision procedures, and so on. More people and investment seem to be badly needed.

In the next chapter, we will show how a UNITY program —and related concepts— can be represented in HOL. We will give examples of a program verification and property refinement. Finally, some main results concerning the round solvability of minimal cost functions and the FSA algorithm will be shown.

HOL is, as said in the Introduction, an interactive theorem prover: one types a formula, and proves it step by step using any primitive strategy provided by HOL. Later, when the proof is completed, the code can be collected and stored in a file, to be given to others for the purpose of re-generating the proved fact, or simply for the documentation purpose in case modifications are required in the future. One of the main strengths of HOL is the availability of a so-called meta language. This is the programming language —which is ML— that underlies HOL. The logic with which we write a formula has its own language, but manipulating formulas and proofs has to be done through ML. ML is a quite powerful language, and with it we can combine HOL primitive strategies to form more sophisticated ones. For example we can construct a strategy which repeatedly breaks up a formula into simpler forms, and then tries to apply a set of strategies, one by one until one that succeeds is found, to each sub-formula. With this feature, it is possible to invent strategies that automate some parts of the proofs.

HOL is however not generally attributed as an automatic theorem prover. Full automation is only possible if the scope of the problems is limited. HOL provides instead a general platform which, if necessary, can be fine-tuned to the application at hand.

HOL abbreviates Higher Order Logic, the logic used by the HOL system. Roughly speaking, it is just like predicate logic with quantifications over functions being allowed. The logic determines the kind of formulas the system can accept as 'well-formed', and which formulas are valid. The logic is quite powerful, and is adequate for most purposes. We can also make new definitions, and the logic is typed. Polymorphic types are to some extend supported. New types, even recursive ones, can be constructed from existing ones.

The major hurdle in using HOL is that it is, after all, still a machine which needs to be told in detail what it to do. When a formula needs to be re-written in a subtle way, for us it is still a rewrite operation, one of the simplest things that there is. For a machine, it needs to know which variables precisely have to be replaced, at which positions they are to be replaced, and by what they should replaced. On the one hand HOL has a whole range of tools to manipulate formulas: some designed for global operations such as replacing all x in a formula with y, and some for finer surgical

	standard notation	**HOL notation**
Denoting types	$x \in A$ or $x : A$	"x:A"
Proposition logic	$\neg p$, true, false	"~p", "T", "F"
	$p \wedge q$, $p \vee q$	"p /\ q", "p \/ q"
	$p \Rightarrow q$	"p ==> q"
Universal quantification	$(\forall x, y :: P)$	"(!x y. P)"
	$(\forall x : P : Q)$	"(!y::P. Q)"
Existential quantification	$(\exists x, y :: P)$	"(?x y. P)"
	$(\exists x : P : Q)$	"(?x::P. Q)"
Function application	$f.x$	"f x"
λ abstraction	$(\lambda x. E)$	"(\x. E)"
Conditional expression	if b then E_1 else E_2	"b => E1 | E2"
Sets	$\{a, b\}$, $\{f.x \mid P.x\}$	"{a,b}", "{f x | P x}"
Set operators	$x \in V$, $U \subseteq V$	"x IN V", "U SUBSET V"
	$U \cup V$, $U \cap V$	"U UNION V", "U INTER V"
	$U \backslash V$	"U DIFF V"
Lists	$a;s$, $s;a$	"CONS a s", "SNOC a s"
	$[a; b; c]$, st	"[a;b;c]", "APPEND s t"

Figure 8.1: *The HOL Notation.*

operations such as replacing an x at a particular position in a formula with something else. On the other hand it does take quite before one gets a sufficient grip on what exactly each tool does, and how to use them effectively. Perhaps, this is one thing that scares some potential users away.

Another problem is the collection of pre-proven facts. Although HOL is probably a theorem prover with the richest collection of facts, compared to the knowledge of a human expert, it is a novice. It may not know, for example, how come a finite lattice is also well-founded, whereas for humans this is obvious. Even simple fact such as $(\forall a, b :: (\exists x :: ax + b \leq x^2))$ may be beyond HOL knowledge. When a fact is unknown, the user will have to prove it himself. Many users complain that their work is slowed down by the necessity to 'teach' HOL various simple mathematical facts. At the moment, various people are working on improving and enriching the HOL library of facts. For example, for the purpose of proving the FSA algorithm we have also produced libraries about well-founded relations, graphs, and lattices.

Having said all these, let us now take a closer at the HOL system.

8.1 Formulas in HOL

Figure 8.1 shows examples of how the standard notation is translated to the HOL notation. As the reader can see, the HOL notation is as close an ASCII notation can be to the standard notation.

Every HOL formula —from now on called HOL *term*— is typed. There are primitive types such as `":bool"` and `":nat"`, which can be combined using type constructors. For example, we can have the product of type A and B: `":A#B"`; functions from A to B: `":A->B"`; lists over A: `":A list"`; and sets over A: `":A set"`. The user does not have to supply complete type information as HOL is equipped with a type inference system. For example, HOL can type the term `"p ==> q"` from the fact that `==>` is a pre-defined operator of type `":bool->bool->bool"`, but it cannot accept `"x = y"` as a term without further typing information. All types in HOL are required to be non-empty. A consequence of this is that defining a sub-type will require a proof of the non-emptiness of the sub-type.

We can have type-variables to denote, as the name implies, arbitrary types. Names denoting type-variables must always be preceded by a `*` like in `*A` or `*B`. Type variables are always assumed to be universally quantified (hence allowing a limited form of polymorphism). For example `"x IN {x:*A}"` is a term stating `x` is an element of the singleton `{x}`, whatever the type of `x` is.

8.2 Theorems and Definitions

A *theorem* is, roughly stated, a HOL term (of type bool) whose correctness has been proven. Theorems can be generated using *rules*. HOL has a small set of primitive rules whose correctness has been checked. Although sophisticated rules can be built from the primitive rules, basically using the primitive rules is the only way a theorem can be generated. Consequently, the correctness of a HOL theorem is guaranteed. More specifically, a theorem has the form:

```
A1; A1; ... |- C
```

where the `Ai`'s are boolean HOL terms representing assumptions and `C` is also boolean HOL term representing the conclusion of the theorem. It is to be interpreted as: if all `Ai`'s are valid, then so is `C`. An example of a theorem is the following:

```
"P 0 /\ (!n. P n ==> P (SUC n)) |- (!n. P n)"
```

which is the induction theorem on natural numbers[1].

As examples of (frequently used) rules are `REWRITE_RULE` and `MATCH_MP`. Given a list of equational theorems, `REWRITE_RULE` tries to rewrite a theorem using the supplied equations. The result is a new theorem. `MATCH_MP` implements the modus ponens principle. Below are some examples of HOL sessions.

```
#DE_MORGAN_THM ;;
|- !t1 t2. (~(t1 /\ t2) = ~t1 \/ ~t2) /\ (~(t1 \/ t2) = ~t1 /\ ~t2)

#th1 ;;
|- ~(p /\ q) \/ q

#REWRITE_RULE [DE_MORGAN_THM] th1 ;;
|- (~p \/ ~q) \/ q
```

[1] All variables which occur free are assumed to be either constants or universally quantified.

The line numbers have been added for our convenience. The `#` is the HOL prompt. Every command is closed by `;;`, after which HOL will return the result. On line 1 we ask HOL to return the value of `DE_MORGAN_THM`. HOL returns on line 2 a theorem, de Morgan's theorem. Line 4 shows a similar query. On line 7 we ask HOL to rewrite theorem `th1` with theorem `DE_MORGAN_THM`. The result is on line 8.

The example below shows an application of the modus ponens principle using the `MATCH_MP` rule.

```
#LESS_ADD ;;
|- !m n. n < m ==> (?p. p + n = m)

#th2 ;;
|- 2 < 3

#MATCH_MP LESS_ADD th2;;
|- ?p. p + 2 = 3
```

As said, in HOL we have access to the programming language ML. HOL terms and theorems are objects in the ML world. Rules are functions that work on these objects. Just as any other ML functions, rules can be composed like `rule1 o rule2`. We can also define a recursive rule:

```
letrec REPEAT_RULE b rule x =
       if b x then REPEAT_RULE b rule (rule x) else x
```

The function `REPEAT_RULE` repeatedly applies the rule `rule` to a theorem `x`, until it yields something that does not satisfy `b`. As can be seen, HOL is highly programmable.

In HOL a *definition* is also a theorem, stating what the object being defined means. Because HOL notation is quite close to the standard mathematical notation, new objects can be, to some extend, defined naturally in HOL. Above it is remarked that the correctness of a HOL theorem is, by construction, always guaranteed. The correctness is however relative to the axioms of HOL. While the latter have been checked thoroughly, one can in HOL introduce additional axioms, and in doing so, one may introduce inconsistency. Therefore adding axioms is a practice generally avoided by HOL users. Instead, people rely on definitions. While it is still possible to define something absurd, we cannot derive false from any definition. Below we show how things can be defined in HOL.

```
#let HOA_DEF = new_definition
   (`HOA_DEF`,
    "HOA (p,a,q) =
       (!(s:*) (t:*). p s /\ a s t ==> q t)") ;;

HOA_DEF = |- !p a q. HOA(p,a,q) = (!s t. p s /\ a s t ==> q t)
```

The example above shows how Hoare triples can be defined (introduced). Here, the limitation of HOL notation begins to show up. We denote a Hoare triple with $\{p\}\ a\ \{q\}$. Or, we may even want to write it like this: $p \xrightarrow{a} q$. A good notation greatly improves the readability of formulas. Unfortunately, at this stage of its development, HOL does not support fancy symbols. Formulas have to be typed linearly from left to right (no stacked symbols or such). Infix operators can be defined, but that is as far as it goes. This is of course not a specific problem of HOL, but of theorem provers in general. If we may quote from Nuprl User's Manual —Nuprl is probably a theorem prover with the best support for notations:

> *In mathematics notation is a crucial issue. Many mathematical developments have heavily depended on the adoption of some clear notation, and mathematics is made much easier to read by judicious choice of notation. However mathematical notation can be rather complex, and as one might want an interactive theorem prover to support more and more notation, so one might attempt to construct cleverer and cleverer parsers. This approach is inherently problematic. One quickly runs into issues of ambiguity.*

Notice that in the above definition, the `s` and the `t` have a polymorphic type of `":*var->*val"`. That is, they are states, functions from variables to values, whatever 'variables' and 'values' may be.

8.3 Theorem Proving in HOL

To prove a conjecture we can start from some known facts, then combine them to deduce new facts, and continue until we obtain the conjecture. Alternatively, we can start from the conjecture, and work backwards by splitting the conjecture into new conjectures, which are hopefully easier to prove. We continue until all conjectures generated can be reduced to known facts. The first yields what is called a *forward proof* and the second yields a *backward proof*. This can illustrated by the tree in Figure 8.2. It is called a proof tree. At the root of the tree is the conjecture. The tree is said to be closed if all leaves are known facts, and hence the conjecture is proven if we can construct a closed proof tree. A forward proof attempts to construct such a tree from bottom to top, and a backward proof from top to bottom.

In HOL, new facts can readily be generated by applying HOL rules to known facts, and that is basically how we do a forward proof in HOL. HOL also supports backward proofs. A conjecture is called a *goal* in HOL. It has the same structure as a theorem:

```
A1; A2; ... ?- C
```

Note that a goal is denoted with `?-` whereas a theorem by `|-`. To manipulate goals we have *tactics*. A tactic may prove a goal —that is, convert it into a theorem. For example `ACCEPT_TAC` proves a goal `?- p` if `p` is a known fact. That is, if we have the theorem `|- p`, which has to be supplied to the tactic. A tactic may also transform a goal into new goals —or *subgoals*, as they are called in HOL—, which hopefully are easier to prove.

Many HOL proofs rely on rewriting and resolution. Rewrite tactics are just like rewrite rules: given a list of equational theorems, they use the equations to rewrite

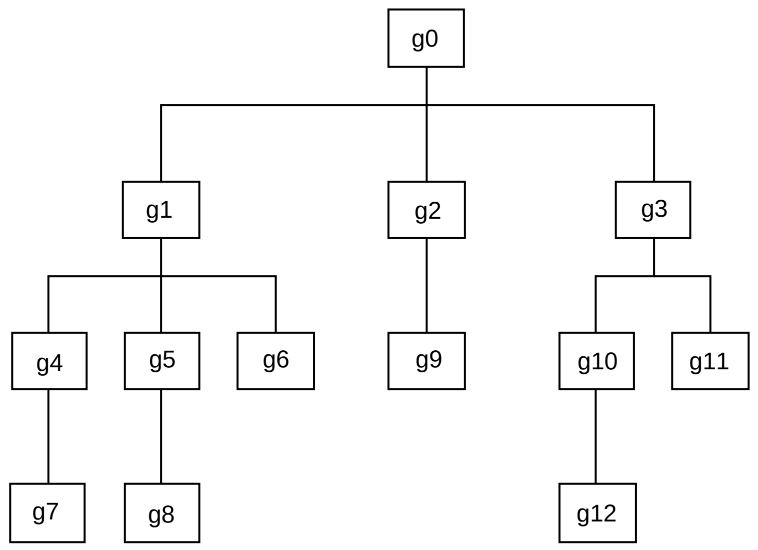


Figure 8.2: *A proof tree.*

the right-hand side of a goal. A resolution tactic, called `RES_TAC`, tries to generate more assumptions by applying, among other things, modus ponens to all matching combinations of the assumptions. So, for example, if `RES_TAC` is applied to the goal:

```
"0<x"; "!y. 0<y ==> z<y+z"; "z<x+z ==> p"   ?-   "p"
```

will yield the following new goal:

```
"z<x+z"; "0<x"; "!y. 0<y ==> z<y+z"; "z<x+z ==> p"   ?-   "p"
```

Applying `RES_TAC` to the above new goal will generate "p" and the tactic will then conclude that the goal is proven, and return the corresponding theorem.

Tactics are not primitives in HOL. They are built from rules. When applied to a goal `?- p`, a tactic generates not only new goals —say, `?- p1` and `?- p2`— but also a justification function. Such a function is a rule, which if applied, in this case, to theorems of the form `|- p1` and `|- p2` will produce `|- p`. When a composition of tactics proves a goal, what it does is basically re-building the corresponding proof tree from the bottom, the known facts, to the top using the generated justification functions to construct new facts along the tree.

HOL provides much better support for backward proofs. For example, HOL provides tactics combinators, also called *tacticals*. For example, if applied to a goal, `tac1 THEN tac2` will apply `tac1` first then `tac2`; `tac1 ORELSE tac2` will try to apply `tac1`, if it fails `tac2` will be attempted; and `REPEAT tac` applies `tac` until it fails. On the other hand, no rules combinators are provided. Of course, using the meta language ML it is quite easy to make rules combinators.

HOL also provides a facility, called the *sub-goal package*, to interactively construct a backward proof. The package will memorize the proof tree and justification functions generated in a proof session. The tree can be displayed, extended, or partly un-done.

```
1 #set_goal ([],"!s. MAP (g:*B->*C) (MAP (f:*A->*B) s) = MAP (g o f) s");;
2 "!s. MAP g(MAP f s) = MAP(g o f)s"
3
4 #expand LIST_INDUCT_TAC ;;
5 2 subgoals
6 "!h. MAP g(MAP f(CONS h s)) = MAP(g o f)(CONS h s)"
7   1  ["MAP g(MAP f s) = MAP(g o f)s" ]
8
9 "MAP g(MAP f[]) = MAP(g o f)[]"
10
11 #expand ( REWRITE_TAC [MAP]);;
12 goal proved
13 |- MAP g(MAP f[]) = MAP(g o f)[]
14
15 Previous subproof:
16 "!h. MAP g(MAP f(CONS h s)) = MAP(g o f)(CONS h s)"
17   1  ["MAP g(MAP f s) = MAP(g o f)s" ]
18
19 #expand (REWRITE_TAC [MAP; o_THM]);;
20"!h. CONS(g(f h))(MAP g(MAP f s)) = CONS(g(f h))(MAP(g o f)s)"
21  1  ["MAP g(MAP f s) = MAP(g o f)s" ]
22
23 #expand (ASM_REWRITE_TAC[]);;
24 goal proved
25 . |- !h. CONS(g(f h))(MAP g(MAP f s)) = CONS(g(f h))(MAP(g o f)s)
26 . |- !h. MAP g(MAP f(CONS h s)) = MAP(g o f)(CONS h s)
27 |- !s. MAP g(MAP f s) = MAP(g o f)s
28
29 Previous subproof:
30 goal proved
```

Figure 8.3: *An example of an interactive backward proof in HOL.*

Whereas interactive forward proofs are also possible in HOL simply by applying rules interactively, HOL provides no facility to automatically record proof histories (proof trees). To prove a goal `A ?- p` with the package, we initiate a proof tree using a function called `set_goal`. The goal to be proven has to be supplied as an argument. The proof tree is extended by applying a tactic. This is done by executing `expand tac` where `tac` is a tactic. If the tactic solves the (sub-) goal, the package will report it, and we will be presented with the next subgoal which still has to be proven. If the tactic does not prove the subgoal, but generates new subgoals, the package will extend the proof tree with these new subgoals. An example is displayed in Figure 8.3.

We will try to prove $g * (f * s) = (g \circ f) * s$ for all lists s, where the map operator $*$ is defined as: $f * [] = []$ and $f * (a; s) = (f.a); (f * s)$. In HOL $f * s$ is denoted by `MAP f s`. The tactic `LIST_INDUCT_TAC` on line 4 applies the list induction principle, splitting the goal according to whether s is empty of not. This results two subgoals listed on lines 6-9. The first subgoal is at the bottom, on line 9, the second on line 6-7. If any subgoal has assumptions they will be listed vertically. For example, the subgoal on lines 6-7 is actually:

```
"MAP g(MAP f s) = MAP(g o f)s"
     ?-  "!h. MAP g(MAP f(CONS h s)) = MAP(g o f)(CONS h s)"
```

The next `expand` on line 11 is applied to first subgoal, the one on line 9. The tactic `REWRITE_TAC [MAP]` attempts to do a rewrite using the definition of `MAP`[2] and succeeds in proving the subgoal. Notice that on line 13 HOL reports back the corresponding theorem it just proven.

Let us now continue with the second subgoal, listed on line 6-7. Since the first subgoal has been proven, this is now the current subgoal. On line 19, we try to rewrite the current subgoal with the definition of `MAP` and a theorem `o_THM` stating that $(g \circ f)x = g(fx)$. This results in the subgoal in line 20-21. On line 23 we try to rewrite the right hand side of the current goal (line 20) with the assumptions (line 21). This proves the goal, as reported by HOL on line 24. On line 29 HOL reports that there are no more subgoals to be proven, and hence we are done. The final theorem is reported on line 27, and can be obtained using a function called `top_thm`. The state of the proof tree at any moment can be printed using `print_state`.

The resulting theorem can be saved, but not the proof itself. Saving the proof is recommended for various reasons. Most importantly, when it needs to be modified, we do not have to re-construct the whole proof. We can collect the applied tactics — manually, or otherwise there are also tools to do this automatically— to form a single piece of code like:

```
let lemma = TAC_PROVE
  (([],"!s. MAP (g:*B->*C) (MAP (f:*A->*B) s) = MAP (g o f) s"),
    LIST_INDUCT_TAC
    THENL
    [ REWRITE_TAC [MAP] ;
      REWRITE_TAC [MAP; o_THM] THEN ASM_REWRITE_TAC ])
```

◀

Sometimes, it is necessary to do forward proving in HOL. For example, there are situations where forward proofs seem very natural, or if we are given a forward proof to verify. It would be nice if HOL provides better support for forward proving. During our research we have also written a package, called the `LEMMA` package, to automatically record the whole proof history of a forward proof. The history consists basically of theorems. It is implemented as a list instead of a tree, however a labelling mechanism makes sure that any part of the history is readily accessible. Just as with the subgoal package, the history can be displayed, extended, or partly un-done. The package also allows comments to be recorded. The theorems in the history can each be proven using ordinary HOL tools such as rules and tactics (that is, the `LEMMA` package is basically only a recording machine). There is a documentation included with the package, else, if the reader is interested, he can find more information in [Pra93].

[2] The name of the theorem defining the constant `MAP` happens to have the same name. These two `MAP`s really refer to different things.

8.3.1 Equational Proofs

A proof style which is overlooked by the standard HOL is the equational proof style. An equational proof has the following format:

$$\begin{array}{ll} & E_0 \\ \sqsubseteq & \{ \text{ hint } \} \\ & E_1 \\ = & \{ \text{ hint } \} \\ & E_2 \\ \sqsubseteq & \{ \text{ hint } \} \\ & \ldots \\ \sqsubseteq & \{ \text{ hint } \} \\ & E_n \end{array}$$

Each relation E_i is related to E_{i+1} by either the relation $\sqsubseteq$ or $=$. The relation $\sqsubseteq$ is usually a transitive relation so that at the end of the derivation, we can conclude that $E_0 \sqsubseteq E_n$ holds. Equational proofs are used very often. In fact, all proofs presented in this thesis are constructed from equational sub-proofs. For low level applications, in which one relies more on automatic proving, proof styles are not that important. When dealing with a proof at a more abstract level, where less automatic support can be expected, and hence one will have to be more resourceful, what one does is usually write the proof on paper using his most favorite style, and then translate it to HOL. Styles such as the equational proof style does not, unfortunately, fit very well in the standard HOL styles (that is, forward proofs using rules and backward proofs using the subgoal package). So, either one has to adjust himself with the HOL styles —which is not very encouraging for newcomers, not to mention that this offends the principle of user friendliness— or we should provide better support.

There are two extension packages that will enable us to write equational proofs in HOL. The first is called the window package, written by Grundy [Gru91], the other is the `DERIVATION` package which we wrote during our research. The window package is more flexible, namely because it is possible to open sub-derivations. The `DERIVATION` package does not support sub-derivations, but it is much easier to use. The structure and presentation of a `DERIVATION` proof also mimics the pen-and-paper style better. The distinctions are perhaps rooted in the different purposes the authors of the packages had in mind. The window package was constructed with the idea of transforming expressions. It is the expressions being manipulated that are the focus of attention. The `DERIVATION` package is written specifically to mimic equational proofs in HOL. Not only the expressions are important, but the whole proof, including comments, and its presentation format.

A proof using the `DERIVATION` package has the following format:

```
ADD_TRANS ("<",LESS_TRANS)

BD "<" "(x*x) + x" ;;

DERIVE ("=:num->num->bool",
  "(x+1)*x",
  `* distributes over +`,
  REWRITE_TAC [RIGHT_ADD_DISTRIB; MULT_LEFT_1 ]) ;;

DERIVE ("<",
  "(x+1)*(x+1)",
  `monotonicity of *`,
  (CONV_TAC o DEPTH_CONV) num_CONV THEN ONCE_REWRITE_TAC [ADD_SYM]
  THEN REWRITE_TAC [ADD; LESS_MULT_MONO; LESS_SUC_REFL]) ;;

DERIVE ("=:num->num->bool",
  "(x*x) + (2*x) +1",
  `* distributes over +`,
  REWRITE_TAC [RIGHT_ADD_DISTRIB; LEFT_ADD_DISTRIB; MULT_LEFT_1; MULT_RIGHT_1;
               TIMES2; ADD_ASSOC]) ;;
```

Figure 8.4: *An example of an equational proof in HOL.*

```
BD Rel E_0 ;;

DERIVE (Rel, E_1, Hint_1, Tac_1) ;;
DERIVE (=  , E_1, Hint_2, Tac_2) ;;
...
DERIVE (Rel, E_n, Hint_n, Tac_n) ;;
```

Notice how the format looks very much like the pen-and-paper format. The only additional components are the `Tac_i` which are tactics required to justify each derivation step. The proof is initialized by the function `BD`. It sets `E_0` as the begin expression in the derivation and the relation `Rel` is to be used as the base of the derivation. This relation has to be transitive. A theorem stating the transitivity of `Rel` has to be explicitly announced using a function `ADD_TRANS`. Every `DERIVE` step, if successful, extends the derivation history with a new expression, related either with `Rel` or equality with the last derived expression. The supplied hint will also be recorded. An example is shown in Figure 8.4[3].

The tactics required to prove the `DERIVE` steps are usually not easy to construct non-interactively. Therefore the package also provides some functions to interactively construct the whole derivation. This is done by calling the sub-goal package. The `DERIVE` package will take care that the right goal that corresponds to the current derivation step —that is, something of the form `?- Rel E_i E_j` or `?- E_i = E_j`— is passed to the subgoal package. The tactic required to prove the derivation step can be constructed using the subgoal package. When the step is proven, the newly derived expression is automatically added to the derivation history.

[3] The proof in Figure 8.4 is not the shortest possible proof, but it will do for the purpose here.

The derivation in Figure 8.4 corresponds to the following derivation:

$$\begin{array}{ll} & x \times x + x \\ = & \quad \{ \times \text{ distributes over } + \} \\ & (x+1) \times x \\ < & \quad \{ \text{ monotonicity of } \times \} \\ & (x+1) \times (x+1) \\ = & \quad \{ \times \text{ distributes over } + \} \\ & x \times x + 2x + 1 \end{array}$$

from which we conclude $x \times x + x < x \times x + 2x + 1$. The derivation generated by a DERIVATION package can be printed at any time using the function DERIVATION. For example, if the code in Figure 8.4 is executed, calling DERIVATION will generate an output like the nicely printed derivation above, except that it is in the ASCII format. The conclusion of the derivation, that is the theorem

```
|- ((x*x) + x) < ((x*x) + (2*x) + 1)
```

can be obtained using a function called ETD (which abbreviates Extract Theorem from Derivation). A complete manual of the package can be found along with the package.

8.4 Automatic Proving

As the higher order logic —the logic that underlies HOL— is not decidable, there exists no decision procedure that can automatically decide the validity of all HOL formulas. However, for limited applications, it is often possible to provide automatic procedures. The standard HOL package is supplied with a library called `arith` written by Boulton [Bou94]. The library contains a decision procedure to decide the validity of a certain subset of arithmetic formulas over natural numbers. The procedure is based on the Presburger natural number arithmetic [Coo72]. Here is an example:

```
1 #set_goal([],"x<(y+z) ==> (y+x) < (z+(2*y))") ;;
2 "x < (y + z) ==> (y + x) < (z + (2 * y))"
3
4 #expand (CONV_TAC ARITH_CONV) ;;
5 goal proved
6 |- x < (y + z) ==> (y + x) < (z + (2 * y))
```

We want to prove $x < y + z \Rightarrow y + x < z + 2y$. So, we set the goal on line 1. The Presburger procedure, ARITH_CONV, is invoked on line 4, and immediately prove the goal.

There is also a library called `taut` to check the validity of a formula from proposition logic. For example, it can be used to automatically prove $p \wedge q \Rightarrow \neg r \vee s = p \wedge q \wedge r \Rightarrow s$, but not to prove more sophisticated formulas from predicate logic, such as

$(\forall x :: P.x) \Rightarrow (\exists x :: P.x)$ (assuming non-empty domain of quantification). There is a library called `faust` written by Schneider, Kropf, and Kumar [SKR91] that provides a decision procedure to check the validity of many formulas from first order predicate logic. The procedure can handle formulas such as $(\forall x :: P.x) \Rightarrow (\exists x :: P.x)$, but not $(\forall P :: (\forall x : x < y : P.x) \Rightarrow P.y)$ because the quantification over P is a second order quantification (no quantification over functions is allowed). Here is an example:

```
 1 #set_goal([], "HOA(p:*->bool,a,q) /\ HOA (r,a,s)
 2               ==>
 3               HOA (p AND r, a, q AND s)") ;;
 4 "HOA(p,a,q) /\ HOA(r,a,s) ==> HOA(p AND r,a,q AND s)"
 5
 6 #expand(REWRITE_TAC [HOA_DEF; AND_DEF] THEN BETA_TAC) ;;
 7 "(!s t. p s /\ a s t ==> q t) /\ (!s t. r s /\ a s t ==> s t) ==>
 8  (!s t. (p s /\ r s) /\ a s t ==> q t /\ s t)"
 9
10 #expand FAUST_TAC ;;
11 goal proved
12 |- (!s t. p s /\ a s t ==> q t) /\ (!s t. r s /\ a s t ==> s t) ==>
13    (!s t. (p s /\ r s) /\ a s t ==> q t /\ s t)
14 |- HOA(p,a,q) /\ HOA(r,a,s) ==> HOA(p AND r,a,q AND s)
```

In the example above, we try to prove one of the Hoare triple basic laws, namely:

$$\frac{\{p\}\ a\ \{q\}\ \wedge\ \{r\}\ s\ \{s\}}{\{p \wedge r\}\ a\ \{q \wedge s\}}$$

The goal is set on line 1-3. On line 6 we unfold the definition of Hoare triple and the predicate level $\wedge$, and obtain a first order predicate logic formula. On line 10 we invoke the decision procedure `FAUST_TAC`, which immediately proves the formula. The final theorem is reported by HOL on line 14.

So, we do have some automatic tools in HOL. Further development is badly required though. The `arith` library cannot, for example, handle multiplication[4] and prove, for example, $(x+1)x < (x+1)(x+1)$. Temporal properties of a program, such as we are dealing with in UNITY, are often expressed in higher order formulas, and hence cannot be handled by `faust`. Early in the Introduction we have mentioned model checking, a method which is widely used to verify the validity of temporal properties of a program. There is ongoing research that aims to integrate model checking tools with HOL[5]. For example, Joyce and Seger have integrated HOL with a model checker called Voss to check the validity of formulas from a simple interval temporal logic [JS93].

[4] In general, natural number arithmetic is not decidable if multiplication is included. So the best we can achieve is a partial decision procedure.

[5] That is, the model checker is implemented as an external program. HOL can invoke it, and then *declare* a theorem from the model checker's result. It would be safer to re-write the model checker within HOL, using exclusively HOL rules and tactics. This way, the correctness of its results is guaranteed. However this is often less efficient, and many people from circuit design —which are influential customers of HOL— are, understandably, quick to reject less efficient product.

Chapter 9

Refinement and Verification in HOL

It will be discussed how programs and program properties are represented in HOL. A small example of program-property refinement and verification will be presented. Finally, the HOL version of the FSA and broadcast algorithms will be shown, along with some of the main results.

As the title of this thesis implies, we try to address the issue of formal program design. In general, we are interested in distributed programs, and in particular, self-stabilizing ones. The use of formal methods has been recognized as a potential tool —perhaps, in the long term also indispensable— to improve the trustworthiness of distributed systems as such systems typically involve complex interactions where intuitive reasoning becomes too dangerous. Formal methods have also been advocated as a means to construct a proof of correctness hand in hand with the construction of the program. This idea appeals us, which is why throughout the examples presented in Part I, more emphasis was given to the design aspect. The trustworthiness that we gain from a formal design can be significantly increased if the design is mechanically verified with a theorem prover. To do so, first of all we need to embed the formal method being used —a programming logic— into the theorem prover.

By embedding a logic into a theorem prover we mean that the theorem prover is extended by all definitions required by the logic, and all basic theorems of the logic should be made available —either by proving them or declaring them as axioms[1]. There are two kinds of embedding: the so-called *deep embedding* and *shallow embedding*. In a deep embedding, a logic is embedded down to the syntax level, whereas in a shallow embedding only the semantic, or model, of the logic needs to be embedded. A deep embedding is more trustworthy, but basically more difficult as we have to take the grammar of well-formed formulas in the logic into account. Alternatively, an external compiler can be constructed to translate syntax-level representations of programs and specifications to formulas at the semantic level. The reader should not imagine something like a Lisp-to-C compiler. Rather, it is a compiler to do straight forward translations, like converting:

```
    IF x<y THEN x:=f.x ELSE x:=0
```

to:

[1] However, adding axioms, as remarked before, is not a recommended practice.

```
    cond (\s. s x < s y) (assign x (\s. f (s x))) (assign x (\s. 0))
```

There are a number of programming logics suitable for designing distributed programs. UNITY is one of them. In Chapter 4 a brief introduction was provided. Subsequently, we have extended it, by the introduction of compositional rules, to allow a modular design of programs. We have also provided various laws for convergence to facilitate reasoning about self-stabilizing systems. In the previous chapter the reader has been briefly introduced to the theorem prover HOL. A question that remains is: how to use HOL to support the formal design of a distributed program? Well, UNITY can be used to design a program from its specification. We have embedded UNITY and almost all extensions discussed in this thesis in HOL. Basically, because the whole UNITY is available in HOL, the derivation can now take place entirely within HOL. Still, if one prefers the flexibility of pencil and paper, then *one can do the derivation by hand first, either in detail or only sketchy, and later verify it with HOL.*

In UNITY, a program is derived by refining its initial specification[2]. Various basic laws to do properties refinement were provided in Chapters 4 and 5. The laws also include compositionality laws, with which we can split a program into smaller components. When the initial specification has been refined to a set of directly verifiable specifications —for example if they are expressed solely in terms of **unless** and **ensures** —, we can try to 'construct' a program satisfying those specifications. This may be quite difficult if we end up with a large number of specifications. It is true that some of the specifications usually give a clear hint as to what kind of actions should or should not be in the program. Besides, during the derivation the designer often makes certain refinement steps motivated by some idea as to how to implement the resulting specifications. Still, we do recommend that the designer exploits the compositionality laws, so that in the end he will have a separate specification for each part of the program, instead of a large set of specifications for the complete program.

An example of property (specification) refinement will be presented in this chapter, but before we come to that, first it will be explained how we represent a UNITY program in HOL. There are some choices, each having consequences on how convenient certain manipulations will be. UNITY itself has been embedded in HOL by Andersen [And92]. There are differences between our embedding of core UNITY with Andersen's. The main difference is that Andersen defines a program simply as a list of actions. Reasoning about compositionality requires that we have information not only of which actions belong to which programs, but also information on which variables belong to which programs, and what their access-modes are. In addition, various extensions of UNITY discussed in this thesis are not available in Andersen's embedding. Both the embedding in [And92] and ours are shallow.

How we represent UNITY programs in HOL will be explained in Section 9.1, and the HOL representation of program properties in Section 9.2. These are not too difficult since the HOL notation, as shown in the previous chapter, is basically quite close to the (standard) hand notation. A small example of property refinement and verification

[2] Another method is to apply program refinement instead of property/specification refinement. This method is beyond the scope of this thesis. See for example [UHK94]

will be provided in Section 9.3. Finally, in Section 9.4 we will briefly show how we define the FSA and the broadcast algorithms in HOL and how some main results about them look like in HOL. Things may be a bit confusing here as the notation and naming convention we used in HOL is somewhat outdated compared to the hand-notation used in this thesis[3]. Apart from that, the reader should be able to recognize the relation between the HOL definitions and theorems with their hand notated counterparts.

9.1 Representing a Program in HOL

In Chapter 4 we have defined a UNITY program as a quadruple (A, J, V_r, V_w) where A is a set of actions, J is a state-predicate describing allowed initial states, V_r is a set of variables intended to be the read variables of the program, and V_w those to be written.

We can represent the universe of all variables in HOL with a polymorphic type `*var` and the universe of all values the variables can take with `*val` . When we have a concrete program, we may want to, for example, use strings to represent variables, and natural numbers as the domain of values. In this case we simply have to instantiate the polymorphic type `*var` and `*val` to `string` and `num`.

In practice, people often want to have programs in which the variables have different types —and, which may include sophisticated types such as functions or trees. That is, we want a multi-typed universe of values. This is possible, albeit not pleasant, as our universe of values is the type `*val` and hence multi-typed values have to be encoded *within* `*val`. For example, if we want both boolean and integer valued program variables, we should define a new type:

```
define_type = `int_bool_DEF` `int_bool = INT int | BOOL bool` ;;
```

The above defines a new type called `int_bool`. A member of this type has the form `INT n` or `BOOL b` where `n` has the type `int` and `b` the type `bool`. Hence, if we instantiate `*val` with this type we will be able to accommodate both bool and integers values. The problem is that all normal operations on integers and bool now have to be lifted to work on this new type, which is quite tedious. There is another way to represent a program in which differently typed variables are easy to represent. But this representation has its own problem too, as we will see later.

Another interesting problem is how to encode, say, an array of variables? We can consider an array variable f as, indeed, a variable whose values are arrays. This will require the type array to be included in `*val`, and then we will have the same problem as described above. There is also a problem if we want to distribute the array among several processes. Each cell in a distributed array may have to be treated as a variable of its own. In this sense, an array is a collection of variables, organized at some meta level as an array. That is, f is represented by `f:*A->*var` where `*A` is the index type of the array. Furthermore it must be required that f is an injective function, and hence

[3] This is not to say that the HOL notation is obsolete. Only the notation that we use on top the standard HOL notation is.

each `f i` will yield a unique variable. If there are several arrays we must also insist that they map to disjoint parts of `*var`, that is, the program is alias free.

The universe of program-states can be represented by `State`:

```
let State = ":*var -> *val" ;;
```

9.1.1 Predicates and Predicate Operators

State-predicates are mapping from program-states to $\mathbb{B}$. The universe of program-states is represented by `Pred`:

```
let Pred = ":^State -> bool" ;;
```

An example is `(\s:^Pred. (s x = f (s y)))` which is a predicate that characterizes those program-states s satisfying $s.x = f.(s.y)$.

We usually and conveniently denote this predicate as, $x = f.y$. This notation is overloaded in several places. Since this kind of overloading is not possible in HOL, basically everything has to be made explicit using λ abstractions as above. Frequently used operators, such as $\neg$, $\wedge$, $\vee$, and so on, can be defined using auxiliary functions:

HOL-definition 9.1.1

```
|- TT            = (\s. T)                  |- FF            = (\s. F)
|- (NOT p)       = (\s. ~p s)               |- (p AND q)     = (\s. p s /\ q s)
|- (p OR q)      = (\s. p s \/ q s)         |- (p IMP q)     = (\s. p s ==> q s)
|- (p EQUAL q)   = (\s. (p s = q s))        |- (!!i::P. Q i) = (\s. (!i::P. Q i s))
|- (??i::P. Q i) = (\s. (?i::P. Q i s))     |-  |== p        = (!s. p s)
```

So, for example the predicate we usually denote by $(x = f.y) \wedge q$ can be denoted by `(\s. s x = f (s y)) AND q` in HOL. Notice that `(!!i::P. Q i)` and `(??i::P. Q i)` above denote $(\forall i : P.i : Q.i)$ and $(\exists i : P.i : Q.i)$ at the predicate level. `|== p` is how we denote $[p]$ (everywhere p) in HOL.

A notion which keeps appearing in the calculational laws given in Chapters 4 and 5 is predicate confinement. A predicate p is said to be confined by a set of variables V if p does not contain any 'meaningful' information about variables outside V (see also page 30):

$$p \in \mathsf{Pred}.V \;=\; (\forall s, t :: (s \restriction V = t \restriction V) \Rightarrow (p.s = p.t))$$

The function projection $\restriction$ is defined on page 23: $(f \restriction V).x = f.x$ if $x \in V$ and $(f \restriction V).x = \aleph$ otherwise. Predicate confinement is defined as follows in HOL:

HOL-definition 9.1.2

```
|- !V A x. (V Pj A)x = (A x => V x | Nov)
|- !A p.  A CONF p = (!s t. (s Pj A = t Pj A) ==> (p s = p t))
```

9.1.2 Actions

We defined an action as a relation[4] on program-states, describing possible transitions the action can make. The universe of actions can be represented by `Action` in HOL:

```
let Action = ":^State -> ^State -> bool" ;;
```

As an example, recall that an assignment $v := E$ can be defined as $(\lambda s, t.\ t.v = E.s) \sqcap (\mathsf{skip} \upharpoonright \{v\}^c)$. This can be defined as follows in HOL:

```
let Assign_DEF = new_definition
  (`Assign_DEF`,
  "(Assign v E):^Action =
   (Update v E) rINTER (SKIP a_Pj (\x:*var. ~(x=v)))") ;;
```

where `Update` is the action $(\lambda s, t.\ t.v = E.s)$, `rINTER` the conjunction (synchronization) operator, `SKIP` the skip action, and `a_Pj` the action-level projection. They are defined as follows in HOL:

HOL-definition 9.1.3

```
|- !v E. Update v E = (\s t. t v = E s)
|- !a b. a rINTER b = (\s t. a s t /\ b s t)
|- SKIP = (\s t. s = t)
|- !a A. a a_Pj A = (\s t. a (s Pj A) (t Pj A))
```
◀

So, an assignment $x := x + 1$ can be represented by `Assign x (\s. (s x) + 1)`.

9.1.3 Programs

The quadruple (A, J, V_r, V_w) representing a UNITY program can now be represented by the product-type:

```
(^Action) set # ^Pred # *var set # *var set
```

However, as HOL is nimbler with predicates than with sets we decided to represent sets with predicates[5] So, instead, we represent a UNITY program —or, to be more precise: objects of type Uprog— as:

```
let Uprog = ":(^Action -> bool) # ^Pred #
              (*var -> bool)    # (*var -> bool)"
```

The destructors **a**, **ini**, **r**, and **w** used to access the components of an Uprog object are called `PROG`, `INIT`, `READ`, and `WRITE` in HOL. The parallel composition $[\!]$ is called `PAR` in HOL:

[4] Some people prefer to use functions instead of relations. If functions are used, then actions are deterministic.

[5] A better set library is under development.

```
 1 let MinDist = new_definition
 2   (`MinDist`,
 3    "MinDist d (V:*node, N) =
 4      ( (??a::(\a. a IN V). (\act. act = Assign (d a a) (\s. 0))) OR
 5        (??a::(\a. a IN V).
 6           (??b::(\b. b IN V /\ ~(a=b)).
 7                 (\act. act = Assign (d a b) (\s:*var->num. MIN {(s (d a b')) + 1 | b' IN (N b)}))))),
 8         TT,
 9        (??a::(\a. a IN V). (??b::(\b. b IN V). (\v. v = d a b))),
10        (??a::(\a. a IN V). (??b::(\b. b IN V). (\v. v = d a b))) )") ;;
```

Figure 9.1: *The HOL definition of the program* MinDist.

HOL-definition 9.1.4

```
|- !P In R W. PROG(P,In,R,W) = P
|- !P In R W. INIT(P,In,R,W) = In
|- !P In R W. READ(P,In,R,W) = R
|- !P In R W. WRITE(P,In,R,W) = W
|- !Pr Qr.
    Pr PAR Qr = (PROG Pr) OR (PROG Qr),(INIT Pr) AND (INIT Qr),
                (READ Pr) OR (READ Qr),(WRITE Pr) OR (WRITE Qr)
```

We will now give an example of the embedding/representation of a UNITY program. Recall the program for computing the simple minimal distance between any pair of nodes in a network (Figure 5.8). For the reader's convenience it is re-displayed below:

prog	MinDist
read	$\{d.a.b \mid a, b \in V\}$
write	$\{d.a.b \mid a, b \in V\}$
init	true
assign	$([\!]a : a \in V : d.a.a := 0)$
$[\!]$	$([\!]a, b : a, b \in V \wedge a \neq b : d.a.b := \mathsf{min}\{d.a.b' + 1 \mid b' \in N.b\})$

In this case the universe of values `*val` is the natural numbers `num`.

The code in Figure 9.1 is the HOL representation of the program above. It defines the constant `MinDist` which has two parameters: a function `d` representing the array d, and a pair `(V,N)` representing a network. These two parameters are kept implicit in the hand-definition of MinDist above. Lines 4-7 defines the set of actions of the program `MinDist d (V,N)`; line 8 defines its initial condition, which is true; and lines 9 and 10 define respectively the sets of read and write variables of the program.

In Chapter 4 we have also stated that in addition to having the type Uprog, a well-formed UNITY program is required to satisfy some conditions given in Subsection 4.2.1. A predicate Unity was defined (page 40) to characterize the set of well-formed (UNITY) programs. It is re-displayed below:

$$\begin{aligned}\mathsf{Unity}.P \;=\;& (\mathbf{a}P \neq \emptyset) \;\wedge\; (\mathbf{w}P \subseteq \mathbf{r}P) \;\wedge\; (\forall a : a \in \mathbf{a}P : \Box_{\mathsf{En}} a) \;\wedge \\ & (\forall a : a \in \mathbf{a}P : (\mathbf{w}P)^{\mathsf{c}} \not\leftarrow a) \;\wedge\; (\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\rightarrow a)\end{aligned}$$

The first and the second condition are obvious. The third states that all actions should be always-enabled (that is, a transition is always possible from any state). The fourth states that variables not declared as write variable of P are ignored by P, hence they cannot be written. The last conjunct states that variables not declared as read variables of P are invisible to P, hence they do not influence P, and hence P does not read from them.

Below, the definition of the operators $\Box_{\mathsf{En}}$, $\not\leftarrow$, and $\not\rightarrow$ is re-displayed. These operators have been discussed in detail in Chapter 3[6].

i. $\Box_{\mathsf{En}} a \;=\; (\forall s :: (\exists t :: a.s.t))$

ii. $V \not\leftarrow a \;=\; (\forall s,t :: a.s.t \Rightarrow (s \restriction V = t \restriction V))$

iii. $V \not\rightarrow a = (\forall s,t,s',t' :: \left(\begin{array}{c} (s \restriction V^{\mathsf{c}} = s' \restriction V^{\mathsf{c}}) \wedge (t \restriction V^{\mathsf{c}} = t' \restriction V^{\mathsf{c}}) \\ \wedge (s' \restriction V = t' \restriction V) \wedge a.s.t \end{array}\right) \Rightarrow a.s'.t')$

The corresponding HOL definition[7]:

HOL-definition 9.1.5

```
|- !A. ALWAYS_ENABLED A = (!s. ?t. A s t)
|- !V A. V IG_BY A = (!s t. A s t ==> (s Pj V = t Pj V))
|- !V A. V INVI A =
       (!s t s' t'.
          (s Pj (NOT V) = s' Pj (NOT V)) /\ (t Pj (NOT V) = t' Pj (NOT V)) /\
          (s' Pj V = t' Pj V) /\ A s t
           ==>
          A s' t')
```

◀

Now, the HOL definition of the predicate Unity:

HOL-definition 9.1.6

```
|- !P In R W. UNITY(P,In,R,W) =  (?A. P A) /\
                                 (!A :: P. ALWAYS_ENABLED A) /\
                                 (!A :: P. (NOT W) IG_BY A) /\
                                 (!x. W x ==> R x) /\
                                 (!A :: P. (NOT R) INVI A)
```

◀

For example, the program shown in Figure 9.1 can be shown to satisfy the predicate UNITY above.

How one represents each component of a program —variables, values, states, and so on— influences how easily certain manipulations on programs can be carried out. We have chosen to represent a state as a function from `*var` to `*val`. There is another way to represent states, namely with tuples. This is for example used by Back and von Wright in their embedding of the Refinement Calculus in HOL [BvW90] and by Långbacka in his embedding of TLA in HOL [Lån94]. As an example, assume that `*var` consists of only three variables `x`, `y`, and `z`. A state in which the values of these variables are 0,1, and 3 is represented, using our representation, by the function:

```
(\v. (v = x => 0 | (v = y => 1 | 3)))
```

[6] See pages 24, 26, and 27 for more details on $\Box_{\mathsf{En}}$, $\not\leftarrow$, and $\not\rightarrow$.

[7] The projection operator `Pj` is defined in Subsection 9.1.1

Using the tuple representation this is represented by the tuple:

```
(0,1,3)
```

Using tuples to represent states, an assignment $x := y + z$ can be represented by:

```
(\(x,y,z) (x',y',z'). (x'=y+z) /\ (y'=y) /\ (z'=z))
```

The tuple representation does look simpler. Representing multi-typed universe of values is also not a problem. For example if `x` and `y` in the above example are $\mathbb{N}$-valued variables and `z` is a $\mathbb{B}$-valued variable, then a state in which these variables has the values of 0,1 and `true` can be represented by the tuple `(0,1,T)` whereas if the function representation is used we will have to make the effort to instantiate `*val` with a general type to accomodate both $\mathbb{N}$ and $\mathbb{B}$ (for example, the type `int_bool` defined earlier on page 181 will do).

The problem is that with this representation it becomes almost impossible to treat program-variables in isolation (as an objects of their own). In fact, they are not objects, but only positions in the tuples. As a consequence, it will be more difficult to define what a shared variable is. If a program P has $\{x, y\}$ as its variables whose values range over $\mathbb{N}$, and if Q has $\{a, b\}$ as its variables whose values also range over $\mathbb{N}$, then both program share the same universe of states, namely $\mathbb{N} \times \mathbb{N}$. If those variables are intended to be distinct then some trick will be required to impose the distinction. Also, we may have a problem when dealing with a variable number of program-variables. For example, the number of variables the program `MinDist` in Figure 9.1 has, depends on how many nodes there are in the network `(V,N)`. Consequently the length of the tuples representing the states of this program also depends on `(V,N)`, and hence it is not fixed. This is a problem in HOL because tuples are required to have a statically determined length, due to its strong typing rules.

9.2 Program Properties in HOL

In the previous section we have given examples of how a UNITY program and its various components can be represented in HOL. At its current development HOL does not support a sophisticated notation interface —so, we have no fancy symbols or such. The formulas do look rather long and un-friendly, but the components are easily recognizable and they are as close as an ASCII notation can get to the hand notation. In this section we will give examples of how properties of a UNITY program can be specified in HOL.

In UNITY, there are two primitives operators to express the property of a program: the **unless** operator to express safety and the **ensures** operator to express progress. Notions such as stable predicates and invariants can be expressed in terms of **unless** . Given a program, properties expressed in these two operators can be directly verified. A more general progress operator is provided by $\mapsto$, which is defined as the smallest transitive and left-disjunctive closure of **ensures** . However, a $\mapsto$ property cannot be directly verified and has to be proven using the UNITY laws of programming. Most of these laws were presented in Chapter 4.

Below is how we define Hoare triples, unless , ensures , and $\circlearrowright$ in HOL.

HOL-definition 9.2.1

```
1 |- !p A q. HOA(p,A,q) = (!s t. p s /\ A s t ==> q t)
2 |- !Pr p q. UNLESS Pr p q = (!A :: PROG Pr. HOA(p AND (NOT q),A,p OR q))
3 |- !Pr p. STABLE Pr p = UNLESS Pr p FF
4 |- !Pr p q. ENSURES Pr p q = UNITY Pr /\
5                              UNLESS Pr p q /\ (?A :: PROG Pr. HOA(p AND (NOT q),A,q))
```

Line 1 defines Hoare triples; line 2 defines unless ; line 3 defines $\circlearrowright$; and lines 4-5 define ensures . Compare them with their hand definition[8][9]:

i. $\{p\}\ a\ \{q\} \;=\; (\forall s,t :: p.s \wedge a.s.t \Rightarrow q.t)$

ii. $_P\vdash p\ \textsf{unless}\ q \;=\; (\forall a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{p \vee q\})$

iii. $_P\vdash \circlearrowright p \;=\; {}_P\vdash p\ \textsf{unless false}$

iv. $_P\vdash p\ \textsf{ensures}\ q \;=\; ({}_P\vdash p\ \textsf{unless}\ q) \;\wedge\; (\exists a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{q\})$

As an example, a property of the program `MinDist` in Figure 9.1 is, expressed in the hand notation, the following:

$$\vdash \circlearrowright (\forall a,b : a,b \in V : d.a.b = \delta.a.b)$$

where $\delta.a.b$ denotes the actual (simple) minimal distance between a and b. Its definition can be found in equations $5.5.2_{101}$ and 5.5.3. Expressed in HOL this is:

```
STABLE (MinDist d (V,N))
       (!!a::(\a. a IN V). (!!b::(\b. b IN V).
             (\s. s (d a b) = Delta (V,N) a b)))
```

where `Delta` corresponds to the function δ. `Delta` has an extra parameter, namely the network `(V,N)`, which is kept implicit in δ.

The general progress operator leads-to ($\mapsto$) is defined as the TDC[10] —the least transitive and left-disjunctive closure— of ensures :

$$_P\vdash p \mapsto q \;=\; \textsf{TDC}.(\lambda rs.\ {}_P\vdash r\ \textsf{ensures}\ s).p.q$$

where TDC is defined as follows:

$$\textsf{TDC}.R.p.q \;=\; (\forall S : R \subseteq S \wedge \textsf{Trans}.S \wedge \textsf{Ldisj}.S : S.p.q)$$

[8] The meaning of these operators has been discussed in Chapters 3 and 4. See pages 31, 42, and 44 for more details on them.

[9] Notice that in the hand definition $_P\vdash p$ ensures q does not explicitly require that P is a UNITY program. It was implicitly assumed that we are talking about UNITY programs —most of the times. This assumption is not crucial for safety laws, but it is for some progress laws. In HOL, one way or another this assumption will have to be made explicit. We did that simply by putting it in the definition of `ENSURES`.

[10] See also Section 4.3 and Subsection 4.4.1 for a detailed discussion on $\mapsto$ and TDC.

We define TDC and $\mapsto$ as follows in HOL:

HOL-definition 9.2.2

```
1 |- !r s. r SUBREL s = (!x y. r x y ==> s x y)
2 |- !r. TRANS r = (!x y z. r x y /\ r y z ==> r x z)
3 |- !U. LDISJ U =  (!W y. (?x. W x) /\ (!x::W. U x y) ==> U (??x::W. x) y)
4 |- !U x y. TDC U x y =  (!X. (SUBREL U X) /\ (TRANS X) /\ (LDISJ X) ==> X x y)
5 |- !Pr. LEADSTO Pr = TDC(ENSURES Pr)
```

◀

Line 1 defines the meaning of being a sub-relation; line 2 defines transitive relations; line 3 defines left-disjunctive relations; line 4 defines the TDC, and finally line 5 defines $\mapsto$. The $\mapsto$ operator is the standard UNITY operator to express progress. Recall that we proposed to replace this operator with a more complicated progress operator $\rightarrowtail$ (the 'reach' operator) in order to facilitate reasoning on program compositions. The $\rightarrowtail$ is defined as the TDC of a 'confined' and 'stabilized' variant of **ensures**. The hand definition of $\rightarrowtail$ is as follows (see also page 54):

$$\begin{aligned} J\ {}_P\!\vdash p\ \mathfrak{ensures}\ q &= p, q \in \mathsf{Pred}.(\mathbf{w}P)\ \wedge\ ({}_P\!\vdash \circlearrowleft J) \wedge ({}_P\!\vdash J \wedge p\ \mathsf{ensures}\ q) \\ (\lambda p, q.\ J\ {}_P\!\vdash p \rightarrowtail q) &= \mathsf{TDC}.(\lambda p, q.\ J\ {}_P\!\vdash p\ \mathfrak{ensures}\ q) \end{aligned}$$

The HOL definition is as follows:

HOL-definition 9.2.3

```
|- !Pr J p q.
   B_ENS Pr J p q =
        ENSURES Pr(p AND J)q /\ STABLE Pr J /\ (WRITE Pr) CONF p /\ (WRITE Pr) CONF q)
|- !Pr J. REACH Pr J = TDC(B_ENS Pr J)
```

◀

Finally, we also have a special operator to express the convergence property, namely $\rightsquigarrow$. The entire Chapter 5 is dedicated to this operator. It is defined as follows:

$$J\ {}_P\!\vdash p \rightsquigarrow q\ =\ q \in \mathsf{Pred}.(\mathbf{w}P)\ \wedge\ (\exists q' :: (J\ {}_P\!\vdash p \rightarrowtail q' \wedge q)\ \wedge\ ({}_P\!\vdash \circlearrowleft(J \wedge q' \wedge q)))$$

It is defined as follows in HOL:

HOL-definition 9.2.4

```
|- !Pr J p q.
  CON Pr J p q =
      (WRITE Pr) CONF q /\ (?q'. REACH Pr J p(q' AND q) /\ STABLE Pr(q' AND (q AND J)))
```

◀

As an example, the program `MinDist` in Figure 9.1 has the following self-stabilizing property —expressed in hand notation:

$$\mathsf{true} \vdash \mathsf{true} \rightsquigarrow (\forall a, b : a, b \in V : d.a.b = \delta.a.b)$$

Stating that the program will converges to a situation where all $d.a.b$'s are equal to the actual minimal distance between a and b. In the HOL notation this will look like:

```
CON (MinDist d (V,N))
    TT
    TT
    (!!a::(\a. a IN V). (!!b::(\b. b IN V). (\s. s (d a b) = Delta (V,N) a b)))
```

```
prog   ABP
read   {wire, Sbit, output, Rbit, ack}
write  {wire, Sbit, output, Rbit, ack}
init   Sbit ≠ ack ∧ Rbit = ack
assign
 (Send)          if Sbit = ack then wire, Sbit := Exp, next.ack
 (Receive)   []  output, Rbit := wire, Sbit
 (ACK)       []  ack := Rbit
```

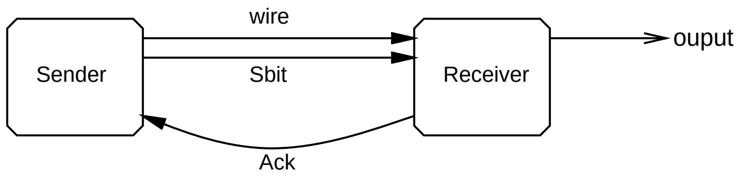


Figure 9.2: *Simple Alternating Bit Protocol*

9.3 An Example

Now that we have shown how things can be defined in HOL, how theorems can be proved, how a UNITY program can be represented in HOL, and how to formulate its properties in HOL, the reader should have some idea how to do refinement or verification with HOL. Basically both boil down to proving theorems. We will here present a small example, just so that the reader will have a more concrete idea about mechanical verification.

Let us consider the program in Figure 9.2. It is a very simple version of an alternating bit protocol. The accompanying picture may be helpful. The sender controls the wire, and basically can assign any value to it through the assignment wire := Exp. The receiver controls the output. For our convenience, on the left column of the assign section we insert the names of the actions (Send, Receive, and ACK). Using the protocol we want to synchronize output with wire so that it satisfies the following specification:

$$(\forall X :: {}_{\mathsf{ALT\_BIT}}\vdash J \wedge (\mathsf{wire} = X) \mapsto (\mathsf{output} = X)) \tag{9.3.1}$$

for some invariant J. To achieve this, the acknowledgement mechanism through Sbit, Rbit, and Ack is used.

We are not going to show full derivation of the program ALT_BIT —besides, it is but a simple program. Part of it will suffice for the purpose of illustration. Let us do some simple calculation on the specification above:

$$J \wedge (\mathsf{wire} = X) \mapsto (\mathsf{output} = X)$$
$$\Leftarrow \quad \{ \mapsto \textsc{Disjunction}_{52} \}$$
$$(J \wedge (\mathsf{Sbit} = \mathsf{Rbit}) \wedge (\mathsf{wire} = X) \mapsto (\mathsf{output} = X)) \;\wedge$$
$$(J \wedge (\mathsf{Sbit} \neq \mathsf{Rbit}) \wedge (\mathsf{wire} = X) \mapsto (\mathsf{output} = X))$$

By $\mapsto$ INTRODUCTION$_{52}$ the last formula above, and hence also (9.3.1), can be refined to:

$$J \wedge (\mathsf{Sbit} = \mathsf{Rbit}) \wedge (\mathsf{wire} = X) \quad \mathsf{ensures} \quad (\mathsf{output} = X) \tag{9.3.2}$$
$$J \wedge (\mathsf{Sbit} \neq \mathsf{Rbit}) \wedge (\mathsf{wire} = X) \quad \mathsf{ensures} \quad (\mathsf{output} = X) \tag{9.3.3}$$

for some invariant J. If we also insist that J is such that $\mathsf{Sbit} = \mathsf{Rbit}$ implies $\mathsf{wire} = \mathsf{output}$ then it follows from $\mathsf{ensures}$ INTRODUCTION$_{47}$ and POST-WEAKENING$_{47}$ that (9.3.2) automatically holds.

Let us now see how the derivation above can be done (verified) in HOL.

Code 9.3.1

```
 1let AB_PROG_lem = prove(
 2   "ENSURES (ALT_BIT Exp Next)
 3           (J1 AND J2 AND (\s:^XState. ~(s `Sbit` = s `Rbit`) /\ (s `wire` = X)))
 4           (\s. s `output` = X)
 5    /\
 6    ENSURES (ALT_BIT Exp Next)
 7           (J1 AND J2 AND (\s. (s `Sbit` = s `Rbit`) /\ (s `wire` = X)))
 8           (\s. s `output` = X)
 9    ==>
10    LEADSTO (ALT_BIT Exp Next)
11           (J1 AND J2 AND (\s. s `wire` = X))
12           (\s. s `output` = X)",
13   STRIP_TAC THEN SUBST1_TAC lemma
14   THEN MATCH_MP_TAC LEADSTO_SIMPLE_DISJ
15   THEN CONJ_TAC THENL
16   [ MATCH_MP_TAC LEADSTO_ENS_LIFT THEN ASM_REWRITE_TAC[] ;
17     MATCH_MP_TAC LEADSTO_ENS_LIFT THEN ASM_REWRITE_TAC[] ])
```

◀

The above will prove a theorem stating that (9.3.2) and (9.3.3) together imply (9.3.1). This theorem —actually, hypothesis— is stated in lines 2-12. Lines 2-4 formulate (9.3.3), lines 6-8 formulate (9.3.2), and lines 10-12 formulate (9.3.1). The previous hand derivation is translated into HOL code in lines 13-17. Compare the following with the hand derivation. Line 14 applies the $\mapsto$ DISJUNCTION. We obtain two specifications. In lines 16 and 17 $\mapsto$ INTRODUCTION is applied to each specification.

So that is an example of doing (verifying) property refinement in HOL. The shown refinement does not lead to a decomposition of the program. For the latter, compositionality laws such as presented in Sections 4.7 and 4.8 are used, but in principle, applying a compositionality law is no different from applying any other theorem.

Let us now see some property verification. Without proof, below is an invariant J that will do for our purpose:

$$((\mathsf{Sbit} = \mathsf{ack}) \Rightarrow (\mathsf{Sbit} = \mathsf{Rbit})) \;\wedge\; ((\mathsf{Sbit} = \mathsf{Rbit}) \Rightarrow (\mathsf{wire} = \mathsf{output})) \tag{9.3.4}$$

Let us now verify that $\mathsf{ALT\_BIT}$ is a well-formed UNITY program, that J is an invariant, and that (9.3.3) indeed holds. Before we can do that, first we need to define $\mathsf{ALT\_BIT}$ in HOL. This is given given below:

HOL-definition 9.3.2

```
|- AB_Rd = [`wire`;`Sbit`;`output`;`Rbit`;`ack`]
|- AB_Wr = [`wire`;`Sbit`;`output`;`Rbit`;`ack`]
|- !Exp Next.
    Send Exp Next =
    (\s. s `Sbit` = s `ack`) THEN ((`wire`,`Sbit`) ASG2 (Exp,(\s. Next(s `Sbit`))))
|- Receive = (`output`,`Rbit`) ASG2 ((\s. s `wire`),(\s. s `Sbit`))
|- ACK = `ack` ASG (\s. s `Rbit`)
|- Init = (\s. ~(s `Sbit` = s `ack`) /\ (s `Rbit` = s `ack`))
|- !Exp Next. ALT_BIT Exp Next = UPROG AB_Rd AB_Wr Init [Send Exp Next;Receive;ACK]
|- J1 = (\s. (s `Sbit` = s `Rbit`) ==> (s `wire` = s `output`))
|- J2 = (\s. (s `Sbit` = s `ack`) ==> (s `Sbit` = s `Rbit`))
```

AB_Rd and AB_Wr are the *lists* of ALT_BIT's read and write variables. Send, Receive, and ACK are the actions of ALT_BIT. The functions THEN, ASG, and ASG2 are the conditional-action construct, the single assignment, and the simultaneous assignment to two variables. Their exact HOL definition is not really important here, but they are as explained in Chapter 3. Init defines the initial condition of ALT_BIT, and ALT_BIT is how we define the program ALT_BIT in HOL. The function UPROG used there is a function that forms an object of type Uprog from its arguments[11]. J1 and J2 are the two conjuncts of the invariant J in (9.3.4).

To prove the well-formedness of a program, we have to check five conditions (see also pages 184). The program is required to consist of at least one action, and its declared write variables should also be declared as read variables. These are easy to check. Then it must be shown that each action is always enabled. This is also easy. It must be shown that no variable not declared as a write variable is written by the program. This can be done by collecting the variables occurring in the left hand sides of the assignments and then comparing them with the set of the declared write variables of the program. Finally, it must be shown that no variable not declared as a read variable will actually influence the program. This may not be easy if we do not do it systematically[12]. This can be done by collecting all variables occurring in the right hand side of assignments, and in the guards of conditionals. Note that we have the constants ASG, ASG2, and THEN which are constructs for actions. These can be considered as defining a language for actions. We define no similar things for expressions (those that may appear at the right hand side of an assignment and as a guard). Consequently, we cannot easily 'collect' the read variables. It does not matter now for we have but a small example. In general though, defining some language for expressions would be handy. So, this does suggest that for the purpose of checking the well-formness of a program, a shallow embedding of UNITY —or of any programming logic, for that matter— is not going to be good enough. The HOL proof of the well-formedness of ALT_BIT is shown below.

[11] Among other things, UPROG has to convert lists of variables into predicates characterizing the membership of the variables.

[12] The read access of a variable is defined in terms of the $\not\leftrightarrow$ operator (see page 185). The operator has a quite complicated definition.

Code 9.3.3

```
 1 let AB_UNITY = prove
 2   ("(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))
 3      ==>
 4      UNITY (ALT_BIT Exp Next)",
 5     STRIP_TAC THEN IMP_RES_TAC INVI_ABS
 6     THEN FIRST_ASSUM (UNDISCH_TAC o concl)
 7     THEN POP_ASSUM (\thm. ALL_TAC)
 8     THEN REWRITE_TAC ALT_BIT_defs THEN UNITY_DECOM_TAC 5) ;;
```

◀

The above will prove that ALT_BIT satisfies the predicate UNITY and hence it is well-formed:

```
|- (NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))
        ==> UNITY (ALT_BIT Exp Next)
```

Except for the read access constraint —that is: $(\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\rightarrow a)$— everything is proven automatically by the tactic UNITY_DECOM_TAC on line 8. The above proof takes about 3 seconds, generating 700 intermediate theorems in the process. The read access constraint is proven by referring to a lemma INVI_ABS on line 5. The lemma itself is proven apart using some smart tactics. It takes about 12 seconds, generating 4400 intermediate theorems. On line 2 we assume that $(\mathbf{r}(\mathsf{ALT\_BIT}))^{\mathsf{c}}$ is invisible to the Send action. This has to be assumed because nothing was said about the expression *Exp* on the right hand side of the assignment in Send —hence we do not know to which variables it may refer.

To prove that J is an invariant we have to show that it is implied by the initial condition of ALT_BIT and that J unless false holds. The HOL proof of the latter is shown below:

Code 9.3.4

```
 1 let AB_INV2 = prove(
 2    "Distinct_Next Next ==> UNLESS (ALT_BIT Exp Next) (J1 AND J2) FF",
 3    REWRITE_TAC ALT_BIT_defs THEN DISCH_TAC
 4    THEN UPROG_UNFOLD_TAC
 5    THEN UNLESS_DECOM_TAC THENL
 7    [ %-- send action --%
 8      COND_CASES_TAC THEN ASM_REWRITE_TAC[] THEN EQ_PROVE_TAC 2 ;
 9      %-- receive actiion  --%
10      EQ_PROVE_TAC 2 ;
11      %-- acknowledgement action --%
12      EQ_PROVE_TAC 2 ]) ;;
```

◀

The above code will prove the following theorem:

```
|- Distinct_Next Next ==> UNLESS (ALT_BIT Exp Next) (J1 AND J2) FF
```

The assumption Distinct_Next Next states that Next x is unequal to x, which is required to prove the above. The tactic UPROG_UNFOLD_TAC in line 4 unfolds a program into its components, and the tactic UNLESS_DECOM_TAC in line 5 unfolds the definition of UNLESS. After the application of these two tactics we obtain a goal expressed in the first order predicate logic. This can be split into three subgoals, one for each action in ALT_BIT. If the reader looks closely at the the definition of J (J1 AND J2) in (9.3.4), it mainly involves equalities. Each subgoal referred above can be proven simply by exploiting the transitivity and symmetry properties of the equality. This is done by the tactic `EQ_PROVE_TAC` in lines 8, 10, and 12.

The proof above takes about 32 seconds and generates 16200 intermediate theorems. As an illustration, the three generated subgoals after executing the steps in lines 3-5 look something like:

```
"(!n. ~(n = Next n)) ==> ((s `Sbit` = s `Rbit`) ==> (s `wire` = s `output`)) /\  ((s `Sbit` = s `ack`) ==>
 (s `Sbit` = s `Rbit`)) ==> (t `wire` = s `wire`) ==> (t `Sbit` = s `Sbit`) ==> (t `output` = s `output`) ==>
 (t `Rbit` = s `Rbit`) ==> (t `ack` = s `Rbit`) ==>  ((t `Sbit` = t `Rbit`) ==> (t `wire` = t `output`)) /\
 ((t `Sbit` = t `ack`) ==> (t `Sbit` = t `Rbit`))"

"(!n. ~(n = Next n)) ==>  ((s `Sbit` = s `Rbit`) ==> (s `wire` = s `output`)) /\ ((s `Sbit` = s `ack`) ==>
 (s `Sbit` = s `Rbit`)) ==>  (t `wire` = s `wire`) ==> (t `Sbit` = s `Sbit`) ==>  (t `output` = s `wire`) ==>
 (t `Rbit` = s `Sbit`) ==> (t `ack` = s `ack`) ==>  ((t `Sbit` = t `Rbit`) ==>
 (t `wire` = t `output`)) /\ ((t `Sbit` = t `ack`) ==> (t `Sbit` = t `Rbit`))"

"(!n. ~(n = Next n)) ==>
 ((s `Sbit` = s `Rbit`) ==> (s `wire` = s `output`)) /\  ((s `Sbit` = s `ack`) ==> (s `Sbit` = s `Rbit`)) ==>
 (t `wire` = ((s `Sbit` = s `ack`) => Exp s | s `wire`)) ==>
 (t `Sbit` = ((s `Sbit` = s `ack`) => Next(s `Sbit`) | s `Sbit`)) ==>
 (t `output` = ((s `Sbit` = s `ack`) => s `output` | s `output`)) ==>
 (t `Rbit` = ((s `Sbit` = s `ack`) => s `Rbit` | s `Rbit`)) ==>
 (t `ack` = ((s `Sbit` = s `ack`) => s `ack` | s `ack`)) ==>
 ((t `Sbit` = t `Rbit`) ==> (t `wire` = t `output`)) /\  ((t `Sbit` = t `ack`) ==> (t `Sbit` = t `Rbit`))"
```

Each subgoal is handled by the tactics on lines 8,10, and 12 (respectively). Alternatively, we can also write a single, smarter tactic which can be applied to all subgoals:

```
((COND_CASES_TAC THEN ASM_REWRITE_TAC[]) ORELSE ALL_TAC)
THEN EQ_PROVE_TAC 2
```

The above attempts to apply a case analysis with `COND_CASES_TAC` first, and then followed by a rewrite with assumptions with `ASM_REWRITE_TAC[]`. If the case analysis fails (the rewrite cannot fail) nothing happens. Subsequently —regardless the success of the case analysis— the tactic `EQ_PROVE_TAC` is invoked. The above can be used to replace lines 7-12 in Code 9.3.4.

To prove (9.3.3), that is, $J \wedge (\mathsf{Sbit} \neq \mathsf{Rbit}) \wedge (\mathsf{wire} = X)$ **ensures** $(\mathsf{output} = X)$, we will have to prove the **unless** part of the above first. This can be done in a very similar way as the proof of J **unless false** in Code 9.3.4.

Code 9.3.5

```
let AB_SAFE1 = prove(
  "Distinct_Next Next ==>
   UNLESS (ALT_BIT Exp Next)
          (J1 AND J2 AND (\s:^XState. ~(s `Sbit` = s `Rbit`) /\ (s `wire` = X)))
          (\s. s `output` = X)",
   REWRITE_TAC ALT_BIT_defs THEN DISCH_TAC
   THEN UPROG_UNFOLD_TAC
   THEN UNLESS_DECOM_TAC THENL
   [ %-- send action --%
     COND_CASES_TAC THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC
     THEN REC_DISJ_TAC (EQ_PROVE_TAC 2) ;
     %-- receive action  --%
     REPEAT STRIP_TAC THEN REC_DISJ_TAC (EQ_PROVE_TAC 2) ;
     %-- acknowledgement action --%
     REPEAT STRIP_TAC THEN REC_DISJ_TAC (EQ_PROVE_TAC 2) ]) ;;
```

◀

The above takes about 50 seconds for HOL to prove, and generates about 28800 intermediate theorems. It results in the following theorem:

```
|- Distinct_Next Next ==> UNLESS (ALT_BIT Exp Next)
                          (J1 AND (J2 AND (\s. ~(s `Sbit` = s `Rbit`) /\ (s `wire` = X))))
                          (\s. s `output` = X)
```

The following code will prove (9.3.3):

Code 9.3.6

```
 1 let AB_ENS1 = prove(
 2    "(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next)) /\ Distinct_Next Next
 3     ==>
 4     ENSURES (ALT_BIT Exp Next)
 5             (J1 AND J2 AND (\s:^XState. ~(s `Sbit` = s `Rbit`) /\ (s `wire` = X)))
 6             (\s. s `output` = X)",
 7     ENSURES_DECOM_TAC "Receive THENL
 8     [ IMP_RES_TAC AB_UNITY ;
 9       IMP_RES_TAC AB_SAFE1 THEN ASM_REWRITE_TAC[] ;
10       REWRITE_TAC [ALT_BIT; UPROG_DEF; PROG; L2P_DEF; MAP; IS_EL] ;
11       DEL_ALL_TAC THEN REWRITE_TAC ALT_BIT_defs
12       THEN (HOA_DECOM_TAC o fst o dest_list o rand o concl) AB_Rd THEN EQ_PROVE_TAC 2 ] ) ;;
```

The above takes only about 5 seconds to prove, mostly because most verification work was already done in proving the **unless** part of (9.3.3). By its definition, $_P\vdash$ p **ensures** q can be proven by showing that P is well-formed, that there exists an action $a \in \mathbf{a}P$ such that $\{p \wedge \neg q\}$ a $\{p \vee q\}$, and that $_P\vdash$ p **unless** q holds. The tactic ENSURES_DECOM_TAC in line 7 is a special tactic that we wrote to split a goal of the form $_P\vdash$ p **ensures** q as described above. It requires one parameter, namely the action a which we think will ensure the described progress[13]. As an illustration, after applying the tactic ENSURES_DECOM_TAC with the action Receive as its parameter in line 7 we will get the following four subgoals from HOL:

```
"HOA
 ((J1 AND (J2 AND (\s. ~(s `Sbit` = s `Rbit`) /\ (s `wire`=X)))) AND (NOT(\s. s `output`=X)),
 F2R Receive,
 (\s. s `output` = X))"
   2  ["(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))" ]
   1  ["Distinct_Next Next" ]

 "PROG(ALT_BIT Exp Next)(F2R Receive)"
   2  ["(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))" ]
   1  ["Distinct_Next Next" ]

"UNLESS (ALT_BIT Exp Next)
        (J1 AND (J2 AND (\s. ~(s `Sbit` = s `Rbit`) /\ (s `wire` = X))))
        (\s. s `output` = X)"
   2  ["(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))" ]
   1  ["Distinct_Next Next" ]

 "UNITY(ALT_BIT Exp Next)"
   2  ["(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))" ]
   1  ["Distinct_Next Next" ]
```

[13] We can gain more automation by letting the tactic ENSURES_DECOM_TAC search for an ensuring action on its own. This is not too difficult to do. One should take into account that this will cost more computing time.

The first subgoal is listed in line 18. It requires that `ALT_BIT` is a well-formed UNITY program. This has been proven before by Code 9.3.3 and the fact is stated by theorem `AB_UNITY`$_{192}$. The second subgoal is listed in line 12-14 and requires the **unless** part of the original **ensures** property to hold. This has been proven by Code 9.3.5 and the fact is stated by theorem `AB_SAFE1`. The third subgoal is in line 8. It requires that `Receive` to be indeed an action of `ALT_BIT`. This is easy to check. Finally, the last subgoal in lines 1-4 states a Hoare triple which the action `Receive` must satisfy. This can be proven using `EQ_PROVE_TAC` again.

9.4 The FSA and Broadcast Algorithms

In the previous section we have briefly shown how refinement and verification of program properties can be done in HOL. Now we want to turn our attention to the FSA and broadcast algorithms presented in Chapters 5 and 6. The total code we produced for the verification is quite large: 120 KB (and another 110 KB for the domain level FSA algorithm). So, it is quite impossible to present it here. We will however show how the programs are formulated in HOL, and how the main theorems look like.

Let us also repeat the caution made earlier. The hand notation used in this thesis is designed for the reader's convenience. The notation we used in our HOL codes is somewhat more cumbersome. While it is easy to adapt the hand notation to one's liking, the same cannot be said for the HOL codes. The reader will find the naming of some variables to be different. Also we were not quite consistent in representing sets. Sometimes we represent them as sets, and sometimes as predicates. The latter dates back to the early development of our UNITY package in HOL. In the beginning we found that HOL can deal better with predicates. In the future, representing sets as sets will be preferable. Fancy notation such as $J\ {}_P\!\vdash p \rightsquigarrow q$ does not directly transfer to HOL, so we have to encode this as something like `CON P J p q`. Also many functions used in the hand notation actually have hidden parameters. In HOL, those parameters have to be mentioned explicitly. Apart from those distracting details mentioned above, the HOL version the formulas presented soon will actually have the same structure as their hand notated counterparts.

Let us now begin with the FSA algorithm.

9.4.1 The FSA Algorithm

The FSA algorithm is given in Figure 5.9 on page 106. Recall that the algorithm is a general algorithm to self-stabilizingly compute the solution of so-called round solvable problems. The definition of round solvability is given in pages 106 or 111.

Recall that the program FSA consists of components FSA.a: FSA $= ([\![a : a \in V :$ FSA.$a)$. Note that the FSA.a's have disjoint sets of variables. In other words, they do not interfere with each other. What we prove in HOL are properties of FSA.a, not of FSA —this does not restrict the result in any way since, as said, the FSA.a's are independent from each other. Since the a can be considered as a constant in FSA.a

$\mathsf{FSA} = ([\!]\, b : b \in V : \mathsf{FSA}.b)$ where $\mathsf{FSA}.b$ is defined as:

prog	$\mathsf{FSA}.b$
read	$\{r.b.b' \mid b' \in V\} \cup \{r.c.b \mid c \in V\} \cup \{d.b\}$
write	$\{r.c.b \mid c \in V\} \cup \{d.b\}$
init	true
assign	$d.b := \Phi.b.(r.b)$
$[\!]$	$([\!]\, c : c \in V \wedge b \in N.c : r.c.b := d.b)$

Figure 9.3: *The FSA Algorithm*

we will simply drop it. For the reader's convenience, the component is redisplayed in Figure 9.3.

Here is how we define the component $\mathsf{FSA}.b$ (of the program in Figure 9.3) in HOL:

HOL-definition 9.4.1

```
|- !Res V N x r a.
    FSA Res (V,N) x r a =
   (CHF ({Comp Res a x r} |_|
        (GSPEC (\b. (Copy a b x r,(b IN V /\ a IN (N b)))))),
    TT,
    CHF ({x a} |_| (GSPEC (\b. (r a b,(b IN V))))
               |_| (GSPEC (\b. (r b a,(b IN V))))),
    CHF ({x a} |_| (GSPEC (\b. (r b a,(b IN V))))))
```

The notation deserves some explanation. `a` and `b` range over the type `*Vert` of nodes (vertices). `V` ranges over sets of nodes and `(V,N)` is intended to be the network. `Comp Res a x r` and `Copy a b x r` are actions[14]. They denote, respectively, the assignments $x.a := Res.a.(r.a)$ and $r.b.a := x.a$. The `s` and `t` range over `State`. `x` and `r` are arrays of variables.

`GSPEC (\x. (f x, P x))` is how we in general denote the set $\{f.x|P.x\}$ in HOL. This is usually pretty printed as `{f x | P x}`. However, the meaning of a set such as $\{a + b \mid 0 < b\}$ can be ambigous as it is not clear whether we mean a to be a bound or a free variable —we usually deduct the status of a from the context. If a is bound then the set can be denoted by `{a+b | 0<b}` in HOL, otherwise we should revert to the `GSPEC` notation as in 9.4.1.

If `V` is a set, `CHF V` is the predicate that characterizes `V`. Recall that we use predicates to represent sets of actions and variables when representing a program, as in the definition of the type `Uprog` on page 183.

In the hand notation, the program in HOL-definition 9.4.1 will look like this:

prog	$\mathsf{FSA}.Res.(V,N).x.r.a$
read	$\{x.a\} \cup \{r.a.b \mid b \in V\} \cup \{r.b.a \mid b \in V\}$

[14] The exact definition of these two actions can be found in Appendix A.

```
write   $\{x.a\} \cup Setr.b.a \mid b \in V$
init    true
assign  $x.a := Res.a.(r.a)$
[]      $([] b : b \in V \wedge a \in N.b : r.b.a := x.a)$
```

which is the same program as the component FSA.b in Figure 9.3 except for some extra parameters —which throughout Chapter 5 were kept implicit— and the renaming of some variables.

The mapping of a function f to a set V, denoted here as $f * V$, which is the set $\{f.x | x \in V\}$, is usually denoted by `IMAGE f V` in HOL. We denote it with `*> f V`. Quantified (UNITY) parallel composition, as in $([] P : P \in V : P)$, is denoted as `gPAR V`. If we define `Net` as follows:

```
|- !P V. Net P V = gPAR (*> P V)
```

we can write the program $\mathsf{FSA} = ([] b : b \in V : \mathsf{FSA}.b)$ as `NET (FSA Res (V,N) x r) V`.

The main result for the FSA algorithm is stated in Theorem 5.5.1$_{111}$. The theorem states that under some conditions, the algorithm will self-stabilize to a situation in which a predicate $\mathsf{ok}^n_{a,b}$ for all rounds n and all pairs of nodes (a,b) holds. The theorem is re-displayed below —it will be adapted for the program FSA in Figure 9.3:

$$\frac{\begin{array}{c}(V,N) \text{ is a network } \wedge\ (V \neq \emptyset)\ \wedge\ V \text{ is finite } \wedge\ (A \neq \emptyset)\ \wedge\ A \text{ is finite} \\ \mathsf{Trans}.\prec\ \wedge\ \prec \text{ is well-founded} \\ (\forall n, F : n \in A : (\forall m, b' : m \prec n \wedge b' \in N.b :\ \mathsf{ok}^m_{b'}.(F.b'))\ \Rightarrow\ \mathsf{ok}^n_b.(\Phi.b.F))\end{array}}{\mathsf{true}\ {}_{\mathsf{FSA}}\!\vdash \mathsf{true} \rightsquigarrow (\forall n, b : n \in A \wedge b \in V : \mathsf{ok}^n_b.(d.b))} \tag{9.4.1}$$

If we now define the following abbreviations:

HOL-definition 9.4.2

```
|- !V OK n x. dataOK V OK n x = (\s. !a. a IN V ==> OK n a x s)
|- !Pr J V A OK x.
    MDC Pr J V A OK x = CON Pr J TT (!!n:: CHF A. dataOK V OK n x)
```

The conclusion of (9.4.1) can now be written in terms of MDC. Now, here is how (9.4.1) looks like in HOL —some variables are named differently:

HOL-theorem 9.4.3

```
|- sWF A R /\ ~(A={}) /\ ~(V={}) /\ TRANS R /\ FINITE A /\ FINITE V /\ GRAPH (V,N) /\
   (!n a. n IN A /\ a IN V ==> Resolve N (A,R) Res ok n a) /\
   Proper (FSA Res (V,N) x r) V /\
   ONE_ONE x /\ ONE_ONE2 r /\ Distinct x r
   ==>
   MDC (Net (FSA Res (V,N) x r) V) TT V A (MSP ok) x
```

`sWF A R` means that `R`, restricted on `A`, is a well-founded relation. `GRAPH (V,N)` means that `(V,N)` is a network. `Proper (FSA Res (V,N) x r) V` means that we assume that for each node `a` in `V`, `FSA Res (V,N) x r a` is a well-formed UNITY program. `ONE_ONE x` and `ONE_ONE2 r` mean that x and r are injective functions, and `Distinct x r` states that $x.a$ and $r.b.c$ are distinct variables[15] [16].

The most important condition from the theorem above is:

```
(!n a. n IN A /\ a IN V ==> Resolve N (A,R) Res ok n a)
```

It states the round solvability of the problem. The predicate `Resolve` is defined as follows:

HOL-definition 9.4.4

```
|- !N A R Res ok n a.
     Resolve N (A,R) Res ok n a =
     (!f. (!m b. m IN A /\ R m n /\ b IN (N a) ==> ok m b (f b))  ==>  ok n a (Res a f))
```

9.4.2 Round Solvability of Minimal Cost Functions

Round solvability is a strong condition, strong enough in some cases to even characterize the original problem. This has been discussed in Chapter 6. In that chapter, the round solvability of minimum-distance-like functions was also thoroughly investigated. Such a function computes the minimal cost of going from one node to another in a network. This cost depends on, for example, the way we define the cost of going through a link —that is, two physically connected nodes— and on the way we 'sum' the cost of the links that were used as we go from the source node to the destination node. The round solvability of such a function is stated by Theorem 6.4.3$_{129}$. We will show how the theorem looks like in HOL.

Recall that a minimum cost function δ is defined as:

$$\delta.U.b = \sqcap\{\varrho.b.s \mid U \xrightarrow{s} b\}$$

where $\varrho.b.s$ is the cost of the path s. That is, the minimal cost of going from a set of nodes U to a node b is the 'minimal' ($\sqcap$) of the cost of the paths from U to b. The cost of a path is defined as follows:

$$\begin{aligned} \varrho.b.[] &= e \\ \varrho.b.(b';s) &= \mathsf{addW}.b'.b.(\varrho.b'.s) \end{aligned}$$

where e is the cost of an empty path —usually it is $\perp$— and addW is a function we use to 'sum' the cost of the links a path goes through. For a more thorough discussion see Section 6.2. The function δ can be generated with the FSA algorithm using the following generator:

$$\varphi.U.b.f = \begin{cases} e & \text{, if } b \in U \\ \sqcap\{\mathsf{addW}.b'.b.(f.b') \mid b' \in N.b\} & \text{, otherwise} \end{cases}$$

[15] The exact (HOL) definition of all these constants can be found in Appendix A.

[16] Recall that the injective and distinct functions are required to represent arrays of variables (see the discussion in page 181).

For the simple notion of minimal cost —see equations $(5.5.2)_{101}$ and $(5.5.3)$— δ is also the fix point of the function $(\lambda f, U, b.\ \varphi.U.b.f)$. In general, Theorems $6.4.5_{132}$ and $6.4.6_{132}$ give conditions under which δ is a unique fix point of its generator.

The above functions are defined as follows in HOL:

HOL-definition 9.4.5

```
|- (!N A b. PATH N A b[] = b IN A) /\
   (!N A b a s. PATH N A b(CONS a s) = a IN (N b) /\ PATH N A a s)
|- !N A b. PATHS N A b = {s | PATH N A b s}
|- (!addW e b. PathCost addW e b[] = e) /\
   (!addW e b a s. PathCost addW e b(CONS a s) = addW a b(PathCost addW e a s))
|- !N r addW e A b. MiCost N r addW e A b = CAP r (*> (PathCost addW e b) (PATHS N A b))
|- !N r addW e f A b.
     GENmc N r addW e f A b = (b IN A => e | CAP r (*> (\a. addW a b (f a)) (N b)))
```
◀

`PATH N U b s` denotes $U \xrightarrow{s} b$ in HOL. It states that s is a path from U to b. Note the parameter `N` which is kept hidden in $U \xrightarrow{s} b$. In fact, all functions defined above have extra parameters which are all kept hidden in the hand notation. `PATHS N U b` denotes the set of all paths from U to b. `PathCost addW e b s` denotes $\varrho.b.s$, `MiCost N r addW e U b` denotes $\delta.U.b$, and `GENmc N r addW e f U b` denotes $\varphi.U.b.f$. In the definition above, `CAP r V` denotes $\sqcap V$ —the greatest lower bound of V— with respect to r and `*> f V` denotes $f * V$ —the map of the function f on the set V.

As said, the round solvability of δ is stated by Theorem $6.4.3_{129}$. This is how the theorem looks like in HOL:

HOL-theorem 9.4.6

```
|- GRAPH(V,N) /\ CAP_PLa r /\ CAP_Closed r /\
   (!a b. CAP_Distr r r (addW a b)) /\ ~(MiCost N r addW e A a = Top r) /\
   (e = Bot r) /\ (!n a b. ~(n = Top r) ==> I_r r n(addW a b n)) /\
   A SUBSET V /\ ~(A = {}) /\ a IN V /\
   (!a0 m. a0 IN (N a) /\ I_r r m n ==> OKmc r m (MiCost N r addW e A) a0 (f a0))
    ==>
   OKmc r n (MiCost N r addW e A) a (GENmc N r addW e f A a)
```
◀

If `r` is a relation on type `*A`, `CAP_PLa r` means that `r` is a (complete) lattice. That is, for every subset `V` of `*A`, the $\sqcap$ of `V`, that is, `CAP r V` exists. `Bot r` and `Top r` denote the $\perp$ and $\top$ of the relation `r` and `I_r r` denotes the relation $r - I$, that is the greatest non-reflexive subrelation of r. Basically, the meaning of a round-solvability theorem such as above still depends on the choice of predicate `OKmc`. For example, if `true` is chosen then the above does not mean anything useful. Instead, `OKmc` should be defined in such a way, so that when the generator `GENmc` is used in the FSA algorithm, upon reaching its goal —which is described in HOL-theorem 9.4.3— the generator will indeed generate `MiCost`. The choice of `OKmc` has been discussed in Chapter 6. For minimum-distance-like functions, it is defined in $(6.4.2)_{126}$. It is defined as follows in HOL:

HOL-definition 9.4.7

```
|- !r n f a X.
     OKmc r n f a X = (r (f a) n ==> (X = f a))  /\  (r n(f a) ==> r n X)
```

9.4.3 The Broadcast Algorithm

Another example that we discussed in Part I is a self-stabilizing broadcast algorithm BC. This is presented in Figure 6.7_{135}. Given a network (V, N) the program BC.U broadcasts the data from the nodes in $U \subseteq V$ to the rest of the network. The program BC.U consists of write-disjoint components BC.$U.b$: BC.$U = ([\![b : b \in V :$ BC.$U.b)$. For the reader's convenience the code of BC.$U.b$ is re-displayed below:

```
prog    BC.$U.b$
read    $\{r.b'.b \mid b' \in V\} \cup \{r.b.c \mid c \in V\} \cup \{g.b \mid b \in U\} \cup \{d.b\}$
write   $\{r.b.c \mid c \in V\} \cup \{d.b\}$
init    true
assign
        $d.b := \begin{cases} (e_1, g.b) & \text{, if } b \in U \\ \sqcap_3\{\rightarrowtail\!\!\!\leftarrowtail.\mathsf{addW}.b'.b.(r.b.b') \mid b' \in N.b\} & \text{, otherwise} \end{cases}$
$[\![$      $([\![ c : b \in N.c : r.c.b := d.b)$
```

where $\sqcap_3$ belongs to $\sqsubseteq_1 \bigotimes \sqsubseteq_2$

The expression at the right hand side of the assignment to $d.b$ can also be written as genBC.$U.b.(r.b)$ for an appropriately defined genBC. Note that the function genBC has the role of a generator. This how it is defined in HOL:

HOL-definition 9.4.8

```
|- !N r1 r2 addW e f g A b.
     GENbc N r1 r2 addW e f g A b =
       (b IN A => (e,g b) |
                   CAP (r1 >>> r2) (*> (\a. >-< r1 r2 addW a b (f a)) (N b)))
```

`r >>> s` is how we denote $r \bigotimes s$, the lexicographic product of r and s, in HOL. `>-<` is how we denote the crab operator $\rightarrowtail\!\!\!\leftarrowtail$ (see page 132 for the definition of $\rightarrowtail\!\!\!\leftarrowtail$). in HOL.

The program BC.$U.b$ is defined as follows in HOL —some variables are renamed:

HOL-definition 9.4.9

```
|- !V N R1 R2 addW e g Org x r a.
    BC (V,N) R1 R2 addW e g Org x r a =
      ( CHF ({Comp_bc N R1 R2 addW e g Org a x r} |_|
             (GSPEC (\b. (Copy a b x r, (b IN V /\ a IN (N b)))))),
        TT,
        CHF({x a,g a} |_| ({r a b | b IN V} |_| {r b a | b IN V})),
        CHF({x a} |_| {r b a | b IN V}))
```

`Comp_bc N R1 R2 addW e g Org a x r` corresponds to the assignment $x.a :=$ genBC.$Org.a.(r.a)$ in the hand notation. `Copy a b x r b` corresponds to the assignment $r.b.a := x.a$. The reader can check for himself that the above program is —except for a number of extra parameters and the renaming of some variables— in fact the same program as the one in the hand notation presented earlier.

The main result of the program BC is stated in Theorem $6.5.5_{136}$. The HOL version of this theorem is displayed below.

HOL-theorem 9.4.10

```
|- ~(A = {}) /\ ~(V = {}) /\ FINITE A /\ FINITE V /\ GRAPH(V,N) /\
   CAP_PLa r1 /\ CAP_PLa r2 /\ CAP_Closed r1 /\ CAP_Closed r2 /\
   (!a b. CAP_Distr r1 r1 (addW a b)) /\
   (!a. a IN V ==> ~(MiCost N r1 addW e Org a = Top r1)) /\
   (!n a b. ~(n = Top r1) ==> I_r r1 n (addW a b n)) /\
   dClosed r1 A /\ ONE_ONE x /\ Distinct x r /\ ONE_ONE2 r /\
   Org SUBSET V /\ ~(Org = {}) /\
   (!s a. J s /\ a IN Org ==> (P o (SND o (s o g)))a) /\
   (e = Bot r1) /\ (ok = (\n a val. OKbc P r1 n (MiCost N r1 addW e Org) a val)) /\
   (Pr = BC (V,N) r1 r2 addW e g Org x r) /\
   (!a. a IN V ==> STABLE (Pr a) J) /\ Proper Pr V
   ==>
   MDC (Net Pr V) J V A (MSP ok) x
```
◀

`OKbc` is defined as Definition $6.5.3_{134}$. That the above implies that upon reaching its goal the program BC.U will indeed have the data in U broadcasted across the network has been discussed before in Section 6.5.

9.4.4 The Domain Level FSA Algorithm

In Chapter 7 we have discussed how we can generalize the FSA algorithm so that it also works for a network of domains. This is useful especially since in practice many networks of computers are organized in domains. Basically the generalization amounts to lifting the actions done by the algorithm at the node level to the domain level. However, since a domain is no longer an indivisible unit as a node is —it consists of nodes— some complications do arise: domain level information which has to be shared by all nodes within a domain will have to be broadcasted across the domain. This is reflected in the proof (of the domain level FSA algorithm): it shows many similarity to that of the ordinary FSA algorithm, except for the broadcasting part.

The domain level FSA algorithm (also called dFSA algorithm) is presented in Figure 7.5_{153}. Its HOL representation is in Figure 9.4.

Lines 1-5 define the component gFSA. `gFSA...B b` corresponds to gFSA.$\underline{\alpha}.\underline{b}.b$ in Figure 7.5_{153}. It consists of only one action, namely `applyGen gen d cp B b` which applies the generator `gen` (denoted as Φ in gFSA.$\underline{\alpha}.\underline{b}.b$). Lines 7-14 defines the component `ccFSA`. `ccFSA...B C c` corresponds to cFSA.$\underline{\alpha}.\underline{b}.\underline{c}.c$ in Figure 7.5_{153}. This component program is defined in terms of the broadcast program (BC). Lines 16-17 defines `cFSA`. `cFSA...B C` corresponds to cFSA.$\underline{\alpha}.\underline{b}.\underline{c}$ in Figure 7.5_{153}. The program is formed from the broadcasting components `ccFSA`. Together they broadcast data from the boder

```
 1 |- gFSA V gen d cp B b =
 2        (CHF{applyGen gen d cp B b},
 3         TT,
 4         CHF((*> (cp B b) V) |_| {d B b}),
 5         CHF{d B b})
 6
 7 |- ccFSA Nd Nn r1 r2 addW_bc e_bc d cp r  B C c =
 8    BC (Nd C, (\c1. {c2 | c2 IN (Nd C) /\ c2 IN (Nn c1)}))
 9        r1 r2 addW_bc e_bc
10        ((d B) o (Sel Nn Nd B))
11        (Border Nn Nd C B)
12        (\c'. cp C c' B)
13        (r C B)
14         c
15
16 |- cFSA Nd Nn r1 r2 addW_bc e_bc d cp r  B C =
17         Net (ccFSA Nd Nn r1 r2 addW_bc e_bc d cp r  B C) (Nd C)
18
19 |- dFSA_Collect Nd Nn (V,N) r1 r2 addW_bc e_bc Gen d cp r =
20        (GSPEC (\(B,b). (gFSA V (Gen (V,N) B) d cp  B b, (B IN V /\ b IN (Nd B)))))
21         |_|
22        (GSPEC(\(B,C). (cFSA Nd Nn r1 r2 addW_bc e_bc d cp r B C, (C IN V /\ B IN (N C)))))
23
24 |- dFSA Nd Nn (V,N) r1 r2 addW_bc e_bc Gen d cp r =
25         flatUprogs (dFSA_Collect Nd Nn (V,N) r1 r2 addW_bc e_bc Gen d cp r)
```

Figure 9.4: *The HOL version of the dFSA algorithm.*

nodes between domains, say, `B` and `C` to all nodes in `C`.

Lines 19-22 defines `dFSA_Collect` which is a set containing all component programs in the dFSA algorithm. Lines 24-25 defines `dFSA`, the dFSA algorithm. `dFSA...` there corresponds to dFSA.$\underline{\alpha}$ in Figure 7.5$_{153}$. It is defined as the `flatUprogs` of `dFSA_Collect`. The first simply combines all programs in the second using the UNITY parallel composition.

Figure 9.5 presents the HOL version of the main theorem on the domain level FSA algorithm presented in 7.1.2. The theorem states the conditions under which the program will satisfy the specification `DA0`, which is the HOL version of DA0 defined in page 143:

$$\mathsf{DA0}:\quad \mathsf{true}\ {}_{\mathsf{dFSA}.\underline{\alpha}}\!\vdash \mathsf{true} \leadsto (\forall n : n \in A : \mathsf{comOk}^n \wedge \mathsf{dataOk}^n)$$

In HOL, `DA0` is defined as follows:

```
|-  DA0 P J Nd (V,N) A ok ccok d r0 =
    CON  P
         J
         TT
         (!!n::(\n. n IN A). (D_dataOK V Nd ok d n) AND (D_comOK V N ccok n))
```

Compare it with the hand definition above, the reader will be able to recognize which part corresponds to which. We will not go further into detailing the definition of `D_dataOK` and `D_comOK`. They can be found in the Appendix. Suffices here to say that they correspond to dataOk and comOk.

Let us now take a closer look at the conditions stated in Theorem 9.4.11. We will briefly explain what they mean (the meaning of most functions has been explained previously in this section).

HOL-theorem 9.4.11

```
 1 |-
 2 sWF A r0 /\ TRANS r0 /\ ~(A={}) /\ FINITE A /\
 3 CAP_PLa r1 /\ CAP_Closed r1 /\ (Top r1) IN (A_bc) /\ dClosed r1 A_bc /\
 4 (e1 = Bot r1) /\ FINITE A_bc /\ CAP_PLa r2 /\ CAP_Closed r2 /\
 5 GRAPH(V,N) /\ ~(V={}) /\ FINITE V /\
 6 (!B. B IN V ==> FINITE (Nd B)) /\ (!B. B IN V ==> ~(Nd B = {})) /\
 7 (!B C c. C IN V /\ B IN (N C) /\ c IN (Nd C) ==>
 8        ~(MiCost (\c1. {c2 | c2 IN (Nd C) /\ c2 IN (Nn c1)})
 9                 r1 addWbc e1 (Border Nn Nd C B) c
10         =
11         Top r1)) /\
12 (!c c'. CAP_Distr r1 r1 (addWbc c c')) /\
13 (!i c c'. ~(i = Top r1) ==> I_r r1 i (addWbc c c' i)) /\
14 (!B n. B IN V /\ n IN A ==> Resolve N (A,r0) (Gen (V,N)) ok n B) /\
15 (!B b. B IN V /\ b IN (Nd B) ==> UNITY (gFSA V (Gen (V,N) B) d cp  B b)) /\
16 (!B C c. C IN V /\ (B:*Dom) IN (N C) /\ c IN (Nd C) ==>
17         UNITY (ccFSA Nd Nn r1 r2 addWbc e1 d cp r B C c)) /\
18 (!B b. B IN V /\ b IN (Nd B) ==> STABLE (gFSA V (Gen (V,N) B) d cp  B b) J) /\
19 (!B C c. C IN V /\ B IN (N C) /\ c IN (Nd C) ==>
20         STABLE (ccFSA Nd Nn r1 r2 addWbc e1 d cp r B C c) J) /\
21 (!B C. C IN V /\ B IN (N C) ==> ~(Border Nn Nd C B ={})) /\
22 (!B1 b1 B2 C1 c1. ~(d B1 b1 = cp C1 c1 B2)) /\
23 (!B1 b1 B2 C1 c1 c2. ~(d B1 b1 = r C1 B2 c1 c2)) /\
24 (!B1 C1 c1 c2 B2 C2 c3. ~(r C1 B1 c1 c2 = cp C2 c3 B2)) /\
25 (!B1 b1 B2 b2. ~((B1,b1)=(B2,b2)) ==> ~(d B1 b1 = d B2 b2)) /\
26 (!B1 C1 c1 B2 C2 c2. ~((B1,C1,c1)=(B2,C2,c2)) ==> ~(cp C1 c1 B1 = cp C2 c2 B2)) /\
27 (!B1 C1 c1 c2 B2 C2 c3 c4.  ~((B1,C1,c1,c2)=(B2,C2,c3,c4)) ==> ~(r C1 B1 c1 c2 = r C2 B2 c3 c4))
28 ==>
29 DAO (dFSA Nd Nn (V,N) r1 r2 addWbc e1 Gen d cp r)
30      J Nd (V,N) A
31      ok (ccOK Nd Nn r1 e1 addWbc ok cp r)
32      d r0
```

Figure 9.5: *The Main Theorem of the Domain Level FSA Algorithm*

`V` is the set of domains, `N` is the neighborhood function at the domain level. `Nd` is the interior function: given a domain `B`, `Nd A` returns the set of all nodes inside the domain. `Nn` is the neighborhood function at the node level. `A` is the domain of the rounds and `ok` is our satisfaction predicate. The goal of the algorithm is to converge to states where the value of `d B b` satisfies `ok n B`, for all domains `B` and nodes `b` inside `B`. We need three relations to prove the correctness of the program: `r0`, `r1`, and `r2`. The last two are only used by the broadcasting part of the program. `A_bc` is the domain of rounds, also to be used by the broadcast components. `addWbc` is a function to add link-weight, just as in the standard FSA algorithm (see the discussion in Subsection 9.4.2). It is also used in the broadcast components. The domain of rounds `A` has to be non-empty, transitive, and well-founded (line 1).

In lines 7-11 it is required that the minimum cost (with respect to the cost function used by the broadcast components) of going from the border nodes between a domain `B` and a domain `C` to any node in `C` is not equal to $\top$. In other words, all nodes in a domain should be reachable from the nodes bordering with that domain (otherwise we cannot do the broadcasting within the domain `C`).

Lines 15-17 requires that all components of the dFSA program are all well-formed

UNITY programs. Actually, this should be verifiable from the definition of `dFSA`, but we did not have the time to do it. This should not be a problem though.

Lines 22-27 simply state that all variables involved are all distinct (no aliasing). The most important condition is stated by line 14, requiring that the problem is round solvable. Notice that this is the same requirement as the one for the ordinary FSA algorithm (see Hol-theorem $9.4.3_{197}$). This emphasizes what has been said before: the domain level FSA algorithm is just an 'upwards' generalization of the FSA algorithm.

9.5 A Closing Note

Translating UNITY and its extension into HOL was fairly easy. Despite its ASCII format, the HOL notation is quite close to the hand notation. Much of our effort in embedding UNITY went into proving the various calculation laws presented in Part I. Some laws such as the $\textsc{Completion}_{52}$ law are quite hard to prove.

Programs with a fixed number of variables and actions such as the program ALT_BIT are quite easy to define in HOL. On the other hand, an abstract algorithm the size of which depends on some parameters tends to get complicated when manipulated in HOL, especially if various sophisticated mathematical structures such as lattices or trees are involved in the parameters of the algorithm, or in the specifications of the algorithm. The FSA and broadcast algorithms, which are quite typical distributed algorithms, are examples of such an algorithm. The examples of HOL definitions and theorems presented in the previous section readily suggests that a notation interface is a great 'must' to make theorem provers more accessible to computer scientists seeking to use the tools to increase the trust-worthiness of their distributed programs.

Our embedding of UNITY in HOL itself, although it serves very well in our experiment, is still not adequate for more practical applications on a routine basis. For such an application we would want, for example, more structuring mechanisms other than the parallel composition —sequential composition is an example thereof; we may also want to have laws that govern communications through channels; we are likely going to need multi-typed universe of values and so on. These areas are all open to further investigation. Each addition of a feature will likely add to the complexity of the (UNITY) logic. It is still an open question how long we can keep adding features before the logic becomes so complicated that we start losing our grip. Surely we can expect that some features can be handled automatically by the computer, but, as said, this is still an untrodden area.

Chapter 10

Concluding Remarks

DURING our four years research, most time and energy was expended on the verification work with the theorem prover HOL. As a beacon, we took the FSA algorithm (Chapter 5) and set the verification of this algorithm as a goal. The path was more or less laid out since a formal proof, written in UNITY, for the algorithm was provided in [Len93]. Basically all that one has to do is to translate the proof to HOL codes. However, we rejected this option, preferring to first re-write the proof in [Len93]. We wanted a more intuitive proof. After all, the philosophy of program design is to be able to go back and forth between the intuition level and the formula level. The programming logic UNITY itself, which we chose as our framework, also needed to be enriched: dealing with complicated formulas with only low level calculational laws at hand is not very practical. Many things can be added: local variables, input variables, loops, exceptions, clocks, procedures, and so on. We are particularly interested in two issues: self-stabilization and compositionality. So we extended UNITY with a notion of stabilization, provided a set of basic properties, verified various compositionality results, and even added laws for write-disjoint parallel composition. These have been discussed in Chapter 4 and Chapter 5. The extension has been mechanically verified in HOL, and the result is available for everyone. Combined with proof tools available in HOL, this will provide an environment for a user to directly verify a program as he designs it using HOL —here, the assumption is of course that one employs a formal method in designing.

We produced lots of theorems from a code of more than 750 KB of proofs. As a comparison, the HOL library of sets contains 90 KB proofs, the library of real numbers, which is quite large, contains 480 KB proofs, and a library of some temporal programming logic by G. Tredoux contains 260 KB proofs. Of the 750 KB we produced, 70 KB deals with the core UNITY —that is, UNITY with unless , ensures and $\mapsto$ as in [CM88]. Twice that much is dedicated to the extensions we talked about above.

A large portion, about a half, of it does not actually have anything to do with program design and verification —that is, not directly. This portion contains proofs for standard mathematical facts, such as those from graph theory or lattice theory, which were not available in HOL. This is a common problem: anyone seriously using a theorem prover is likely to have experienced it. As it stands, theorem prover is a new technology. People are still working on it, improving it, and together building the library of facts. As the library gets larger and larger, new problems can be expected to arise. The notation will need to be standardized, good documentation will be needed, and there should be some kind of a dictionary of mathematical facts. As people work on the technology in various institutions across the world, it will be difficult to couple them in a tight cooperation. That way as it is now, everyone who ever worked on embedding some programming logic in HOL seems to have written his own version of a CPO library, his own predicate library, his own weakest pre-condition library, and so on. This is definitely not the way to do it. Just like an encyclopedia, the whole library of a theorem prover has to be well organized.

In addition to the above 750 KB, another 120 KB was spent to prove the FSA algorithm and 110 KB to prove the domain level version of it. The algorithm is thus verified. If we learn anything from the the experiment, it is the blowing up of the hand proof as provided in Chapters 5 and 7 to a 230 KB piece of code. This is mainly to be blamed to the problem itself. To make the FSA algorithm as general as possible, we parameterized it with various things: the network, the generator, the function to be computed, a domain of rounds, and so on. The proof will involve manipulation of these parameters. This has been omitted from the hand proof in Chapters 5 and 7 because we consider it as less interesting. In the world of mechanical theorem proving we have to work in detail, and it is quite possible that details overwhelm the principle. As many abstract distributed algorithms are also parameterized with a whole range of parameters, one can expect a similar scaling up. By this remark we do not mean to discourage the reader from attempting mechanical verification. On the contrary, we urge him. It is just that we feel we should warn a new user of what lies awaiting for him.

Finally, we certainly hope that others can now benefit from the extended UNITY package that we wrote and use it to verify distributed programs. Such an endeavor may cost some work, especially since abstract algorithms often involve higher order formulas which are likely cannot be automatically verified. We believe however that such work still lies within the limits of practicality.

Appendix A

HOL Definitions and Theorems

THIS appendix provides lists of definitions and verified theorems as they are in HOL. They concern the core UNITY, the extension thereof as discussed in Chapter 4, the convergence operator as discussed in Chapter 5, the FSA algorithm presented in Chapter 5, and the round solvability of minimum-distance-like functions as presented in Chapter 6. Not all definitions and theorems that we produced will be listed —it would take too much space. Only those we consider relevant or interesting for the reader will be included. As warned in the Introduction, there will be a discrepancy between the HOL definitions and theorems and the 'hand' versions as presented in earlier chapters. Not only that the HOL notation differs from the standard hand notation, but objects may be called by other names and the order of arguments of a function may be different. In Chapter 9 we have explained how *some* major definitions and theorems relate with their HOL version, but in general the reader can always try to go over the lists here and find out that the discrepancy between the HOL version and the hand version of the definitions and the theorems is not really that big.

In the sequel IN, UNION, and INTER are HOL names for the set operators $\in$, $\cup$, and $\cap$. |_| is sometimes used instead of UNION.

A.1 Standard Operators on Relations

```
SUBREL: % sub-relation relation %
  |- !r s. r SUBREL s = (!x y. r x y ==> s x y)

rUNION: % union of two relations %
  |- !r s. r rUNION s  =  (\x y. r x y \/ s x y)

rINTER: % intersection of two relations %
  |- !r s. r rINTER s  =  (\x y. r x y /\ s x y)

rSEQ: % sequential compositon of relations %
  |- !R1 R2.
     R1 rSEQ R2 = (\x y. (?y. R1 x y /\ R2 y z))

rId: % the identity relation %
  |- rId = (\x y. (x=y))

rBOT: % the empty relation (bottom) %
  |- rBOT = (\x y. F)

rTOP: % the full relation (top) %
  |- rTOP = (\x y. T)

TRANS_DEF:
  |- !r. TRANS r =
     (!x y z. r x y /\ r y z ==> r x z)

REFL_DEF:
  |- r. REFL r = (!x. r x x)

ANTI_REFL:
  |- !r. ANTI_REFL r = (!x. ~r x x)

ANTI_SYM:
  |- ANTI_SYM r =
     (!x y. r x y /\ r y x ==> (x=y))

REL_LEXII: % lexicographic product of
             two relations %
  |- !R1 R2 x y.
      (R1 >>> R2)x y =
          R1(FST x)(FST y) /\ ~(FST x = FST y) \/
          (FST x = FST y)  /\ R2(SND x)(SND y)

I_remove: % the largest non-reflexive
             sub-relation $
  |- !r a b. I_r r a b = r a b /\ ~(a = b)
```

A.2 Predicate Operators

Below are the definition of the boolean operators ¬, ∧, ∀, ∃, and so on, lifted to the predicate level.

```
pSEQ_DEF: % everywehere operator %
 |- !p. |== p = (!s. p s)

RES_qOR:
 |- !W P. (??i::W. P i) = (\s. ?i. W i /\ P i s)

RES_qAND:
 |- !W P. (!!i::W. P i) = (\s. !i. W i ==> P i s)

EQUAL_DEF:
  |- !p q. p EQUAL q = (\s. p s = q s)

pIMP_DEF:
  |- !p q. p IMP q = (\s. p s ==> q s)

pOR_DEF:
  |- !p q. p OR q = (\s. p s \/ q s)

pAND_DEF:
  |- !p q. p AND q = (\s. p s /\ q s)

pNOT_DEF:
  |- !p. NOT p = (\s. ~p s)

FF_DEF:
  |- FF = (\s. F)

TT_DEF:
  |- TT = (\s. T)

IMPBY:
  |- !p q. p <== q = q ==> p
```

A.3 Well-Founded Relations

Here are some definitions and theorems about well-founded relations which are relevant for latter.

```
sWF : % well-founded relation %
|- !V R. sWF V R =
           (!A. (!y. y IN V /\ y IN A ==> (?x. x IN V /\ R x y /\ x IN A))
                 ==>
                (V INTER A = {}))

ADMIT_sWF_Ind: % well-founded induction %
  |- !V R.
     ADMIT_sWF_Ind V R =
        (!P. (!y. y IN V /\ (!x. x IN V /\ R x y ==> P x)  ==> P y)
             ==>
             (!x. x IN V ==> P x))

ADMIT_WF_INDUCTION: % also well-founded induction %
  |- !R. ADMIT_WF_INDUCTION R =
           (!P. (!y. (!x. R x y ==> P x) ==> P y) ==> (!x. P x))
```

```
ADMIT_sWF_WF:
  |- !R. ADMIT_WF_INDUCTION R = ADMIT_sWF_Ind UNIV R
```

```
sWF_EQU_IND:
  % well-foundedness and admitance of well-founded induction are equivalent %
  |- !V R. sWF V R = ADMIT_sWF_Ind V R

FIN_ADMIT_sWF_Ind:
  |- !V R. TRANS R /\ ANTI_REFL R /\ FINITE V  ==> ADMIT_sWF_Ind V R
```

A.4 Transitive and Disjunctive Closure

Below, the definition of the smallest transitive and left-disjunctive closure of a relation is given, along with its general properties.

```
LDISJ_DEF: % define left-disjunctive relation %
  |- !U. LDISJ U =
        !W y. (?x. W x) /\ (!x::W. U x y)
              ==>
              U (??x::W. x) y

DISJ_DEF: % define generally disjunctive
            relation %
  |- !W U.
     DISJ W U =
     (!f g. (?i. W i) /\ (!i::W. U (f i) (g i))
            ==>
            U (??i::W. f i) (??i::W. g i))

TDC_DEF: % define smalles transitive and
           left-disjunctive closure %
  |- !U x y.
     TDC U x y =
     (!X. (SUBREL U X) /\ (TRANS X) /\ (LDISJ X)
          ==>
          X x y)

CANCEL: % general cancelation law %
  |- !U p q r s.
     TRANS U  /\ DISJ TT U /\
         U q q /\ U p (q OR r) /\ U r s
         ==>
         U p (q OR s)

BOUNDED_REACH_: % bounded reach induction
                  principle %
  |- ADMIT_WF_INDUCTION LESS /\
     TRANS U /\ LDISJ U /\ U q q /\
     (!m. U (p AND (\s. M s = m))
            ((p AND (\s. LESS (M s) m)) OR q))
     ==>
     U p q

BOUNDED_ALWAYS_REACH_i :
  |- ADMIT_WF_INDUCTION LESS /\
     TRANS U /\ LDISJ U /\ U p p /\
     (!m. U ((NOT p) AND (\s. M s = m))
            ((\s. LESS (M s) m) OR p))
    ==>
    U TT p

TDC_LIFT:
  |- !U. SUBREL U (TDC U)

TDC_LDISJ:
  |- !U. LDISJ (TDC U)

TDC_INDUCT1: % the TDC induction principle %
  |- !U X. SUBREL U X /\ TRANS X /\ LDISJ X
           ==>
          (TDC U) SUBREL X

TDC_SUBREL:
  |- !U V. (U SUBREL V) ==> (U SUBREL (TDC V))

TDC_MONO:
  |- !U V.
     (U SUBREL V) ==> (TDC U) SUBREL (TDC V)
```

A.5 Variables and Actions

Below is the definition of skip, miracle, assignment, conditional, and so on. The notion of predicate confinement, ignored variables, and invisible variables will also be defined. A list of some basic properties follows. These properties were discussed in Chapter 3.

```
Pj_DEF: % projection on functions %
  |- !V A x. (V Pj A)x = (A x => V x | Nov)

p_Pj_DEF: % projection of state predicates %
  |- !p A. p_Pj p A = (\s. p (s Pj A))

a_Pj_DEF: % projection of actions %
  |- !a A.
     a_Pj a A = (\s t. a (s Pj A) (t Pj A))
```

```
SKIP_DEF:
  |- SKIP = rId

MIRA_DEF: % miracle %
  |- MIRA = rBOT

CHAOS_DEF:
  |- CHAOS = rTOP

Update_DEF:
  |- !v E. Update v E = (\s t. t v = E s)

Assign_DEF: % assignment %
  |- !v E.
     Assign v E =
    (Update v E) rINTER (SKIP a_Pj (\x. ~(x=v)))

Cond_DEF % conditional action %
  |- !g a b.
     Cond g a b =
       (\s t. (g s ==> a s t) /\ (~g s ==> b s t))

aREF: % action refinement %
  |- !r s. r aREF s = s SUBREL r

CONF_DEF: % predicate confinement %
  |- !A p.
     A CONF p
     =
     (!s t. (s Pj A = t Pj A) ==> (p s = p t))

ALWAYS_ENABLED: % always enabled action %
  |-  !a. ALWAYS_ENABLED a = (!s. ?t. a s t)

IG_BY_DEF: % ignored variables %
  |- !V A.
     V IG_BY A
     =
     (!s t. A s t ==> (s Pj V = t Pj V))

INVI_DEF: % invisible variables %
  |- !V A.
      V INVI A =
      (!s t s' t'.
        (s Pj (NOT V) = s' Pj (NOT V)) /\
        (t Pj (NOT V) = t' Pj (NOT V)) /\
        (s' Pj V = t' Pj V) /\
        A s t ==>
        A s' t')

HOA_DEF: % Hoare triple %
  |- !p A q.
     HOA(p,A,q) = (!s t. p s /\ A s t ==> q t)
```

```
CONF_ADEF:
  |- V CONF p = (p = p p_Pj V)

CONF_MONO:
  |- !A B p.
     |==(A IMP B) /\ A CONF p ==> B CONF p

CONF_qOR:
  |- !V W P.
     (?i. W i) /\ (!i :: W. V CONF (P i))
     ==>
     V CONF (?? i :: W. P i)

CONF_qAND:
  |- !V W P.
     (!i :: W. V CONF (P i))
     ==>
     V CONF (!! i :: W. P i)

CONF_IMP:
  |- !V p q.
     V CONF p /\ V CONF q
     ==>
     V CONF (p IMP q)

CONF_OR:
  |- !V p q.
     V CONF p /\ V CONF q
     ==>
     V CONF (p OR q)

CONF_AND:
  |- !V p q.
     V CONF p /\ V CONF q
     ==>
     V CONF (p AND q)

CONF_NOT:
  |- !V p. V CONF p ==> V CONF (NOT p)

CONF_FF:
  |- !V. V CONF FF

CONF_TT:
  |- !V. V CONF TT

IG_BY_ADEF:
  |- V IG_BY a = (SKIP a_Pj V) aREF a

IG_BY_MONO:
  |- !A V W.
     |==(V IMP W) /\ W IG_BY A ==> V IG_BY A

INVI_ADEF:
  |- V INVI a
     =
     ((a aREF ((a a_Pj (NOT V))
              rINTER (SKIP a_Pj V)))
     /\
     ((a a_Pj (NOT V)) aREF a))

INVI_MONO:
  |- !A V W.
     |==(V IMP W) /\ W IG_BY A /\ W INVI A
     ==>
     V INVI A

HOA_REF:
  |- a aREF b /\ HOA(p,a,q) ==> HOA(p,b,q)

HOA_Pj_MAP:
  |- HOA (p,a,q)
      ==>
     HOA(p p_Pj V, a a_Pj V, q p_Pj V)

HOA_INVI:
  |- (NOT V) INVI a /\ HOA(p,a,q)
      ==>
     HOA(p p_Pj V, a, q p_Pj V)

HOA_ADD_FRESH:
  |- V IG_BY b /\ (NOT V) INVI a /\
     HOA(p,a,q)
     ==>
     HOA (p p_Pj V, b rSEQ a, q p_Pj V)

Action_Wr_Pred:
  |- (NOT V) IG_BY a /\ W CONF p
      ==>
      HOA ((\s. s Pj (V AND W) =
                f Pj (V AND W)) AND p,
           a,
           (\s. s Pj (V AND W) =
                f Pj (V AND W)) IMP p)

Action_Wr_Pred_Cor:
  |- V IG_BY a /\ V CONF p ==> HOA(p,a,p)
```

A.6 Core UNITY

Below are the definition of the predicate Unity, defining the well-formedness of a UNITY program, and the definitions of all basic UNITY operators.

```
UNLESS:
  |- !Pr p q.
     UNLESS Pr p q =
     (!A :: PROG Pr. HOA(p AND (NOT q),A,p OR q))

ENSURES:
  |- !Pr p q.
     ENSURES Pr p q =
     UNITY Pr /\
     UNLESS Pr p q /\
     (?A :: PROG Pr. HOA(p AND (NOT q),A,q))

LEADSTO:
  |- !Pr. LEADSTO Pr = TDC(ENSURES Pr)

STABLE:
  |- !Pr p. STABLE Pr p = UNLESS Pr p FF

Inv: % strong invariant %
  |- !Pr J.
     Inv Pr J =
     |==((INIT Pr) IMP J) /\ UNLESS Pr J FF

FPp_DEF: % fix point of a program %
  |- !Pr p.
     FPp Pr p =
    (!a s t. PROG Pr a /\ p s /\ a s t
             ==>
             (s = t))

PROG: % the action set of a program %
  |- !P In R W. PROG(P,In,R,W) = P

INP:  % the input variables set of a program %
  |- !Pr. INP Pr = (NOT(WRITE Pr)) AND (READ Pr)

WRITE: % the write variables set of a program %
  |- !P In R W. WRITE(P,In,R,W) = W

READ:  % the read variables set of a program %
  |- !P In R W. READ(P,In,R,W) = R

INIT: % the initial condition of a program %
  |- !P In R W. INIT(P,In,R,W) = In

UNITY: % define a UNITY program %
  |- !P In R W.
     UNITY(P,In,R,W) =
     (?A. P A) /\
     (!A :: P. ALWAYS_ENABLED A) /\
     (!A :: P. (NOT W) IG_BY A) /\
     (!x. W x ==> R x) /\
     (!A :: P. (NOT R) INVI A)
```

A.7 Semantics of UNITY

Below is the definition of the trace semantics of UNITY programs. Various basic temporal operators are given, and an operational meaning of the unless and leads-to operator is given. Some soundness results are also given —these were discussed in Section 4.10.

```
EXEC_DEF: % define fair executions of a program %
  |- !P Sig.
     EXEC P Sig =
     (!i. PROG P (Sig i)) /\
     (!i a. PROG P a ==>
            (?j. i <= j /\ (Sig j=a)))

TRACE_DEF: % define trace semantics of a prog. %
  |- !P Tau.
     TRACE P Tau =
     INIT P (Tau 0) /\
     (?Sig. EXEC P Sig /\
            (!i. Sig i (Tau i) (Tau (SUC i))))

SATISFACTION: % define when a program P satisfies
                a temporal property tp %
  |- !P tp.
     |--- P tp = !Tau. TRACE P Tau ==> tp Tau

NOW_DEF: % temporal operator ’now’%
  |- !p Tau. NOW p Tau = p (Tau 0)

DRop_DEF:
  |- !m f. DRop m f = (\n. f (n+m))

NEXT_DEF: % temporal operator ’next’ %
  |- !tp Tau. NEXT tp Tau = tp (DRop 1 Tau)
```

```
ALWAYS_DEF: % temporal operator ’always’ %
  |- !tp Tau.
     ALWAYS tp Tau = !i. tp (DRop i Tau)

EV_DEF: % temporal operatow ’eventually’ %
  |- tp Tau. EV tp Tau = ?i. tp (DRop i Tau)

tUNLESS_DEF: % operational meaning of ’unless’ %
  |- !tp tq.
     tUNLESS tp tq =
     ALWAYS
     ((tp AND (NOT tq)) IMP (NEXT (tp OR tq)))

tLEADSTO_DEF: % operational meaning of ‘leads to‘
  |- !tp tq.
     tLEADSTO tp tq = ALWAYS (tp IMP (EV tq))
```

```
UNLESS_IMP_tUNLESS: % soundness result for unless %
  |- UNLESS P p q ==> (P |--- tUNLESS (NOW p) (NOW q))

LEADSTO_IMP_tLEADSTO: % soundness result for leads-to %
  |- LEADSTO P p q ==> (P |--- tLEADSTO (NOW p) (NOW q))
```

A.8 Sander’s Subscripted UNITY

Below we give the definitions of Sander’s subscripted UNITY operators and the resulting Substitution Rules. The definitions were originally given in [San91], which were faulty, and repaired in [Pra94]. Sander’s subscripted UNITY were briefly discussed in Section 4.6.

```
UNL: % subscripted unless operator %
  |- !Pr J p q.
     UNL Pr J p q =
     UNLESS Pr(p AND J)(q AND J) /\ Inv Pr J

ENS: % subscripted ensures operator %
  |- !Pr J p q.
     ENS Pr J p q =
    ENSURES Pr(p AND J)(q AND J) /\ Inv Pr J

LTO: % subscripted leads-to operator %
  |- !Pr J. LTO Pr J = TDC(ENS Pr J)

UNL_SUBST: % substitution rule for unless %
  |- !Pr J p q r s.
     |== (J IMP (p EQUAL r)) /\
     |== (J IMP (q EQUAL s)) /\
     UNL Pr J p q
     ==>
     UNL Pr J r s

ENS_SUBST: % substitution rule for ensures %
  |- !Pr J p q r s.
     |== (J IMP (p EQUAL r)) /\
     |== (J IMP (q EQUAL s)) /\
     ENS Pr J p q
     ==>
     ENS Pr J r s

LTO_SUBST: % substitution rule for leads-to %
  |- !Pr J p q r s.
     |== ((J AND r) IMP p) /\
     |== ((J AND q) IMP s) /\
     LTO Pr J p q
     ==>
     LTO Pr J r s
```

A.9 the $\rightarrowtail$ operator

Below the HOL definition of the reach operator, $\rightarrowtail$, is given. A list of its basic properties follows. Most of these properties were listed in Chapter 4.

```
B_ENS:
  |- !Pr J p q. B_ENS Pr J p q = ENSURES Pr(p AND J)q /\
                                 STABLE Pr J /\
                                 (WRITE Pr) CONF p /\
                                 (WRITE Pr) CONF q)

REACH:
  |- !Pr J. REACH Pr J = TDC(B_ENS Pr J)
```

```
REACH_ENS_LIFT:
  |- !Pr J. (B_ENS Pr J) SUBREL (REACH Pr J)

REACH_IMP_gLIFT:
  |- !Pr J p q.
     UNITY Pr /\ STABLE Pr J /\
     (WRITE Pr) CONF p /\ (WRITE Pr) CONF q /\
     |==((p AND J) IMP q)
     ==>
     REACH Pr J p q);

REACH_TRANS:
  |- !Pr J. TRANS(REACH Pr J)

REACH_LDISJ:
  |- !Pr J. LDISJ(REACH Pr J)

REACH_DISJ:
  |- !W Pr J. DISJ W(REACH Pr J)

REACH_INDUCT1:
  |- !X Pr J.
     (B_ENS Pr J) SUBREL X /\ TRANS X /\ LDISJ X
     ==>
     (!p q. REACH Pr J p q ==> X p q)

REACH_REFL:
  |- !Pr J p.
     UNITY Pr /\ STABLE Pr J /\
     (WRITE Pr) CONF p
     ==> REACH Pr J p p);

REACH_gMONO:
  |- !Pr J p q r.
     (WRITE Pr) CONF r /\ REACH Pr J p q /\
     |==((q AND J) IMP r)
     ==>
     REACH Pr J p r

REACH_gANTIMONO:
  |- !Pr J p q r.
     (WRITE Pr) CONF p /\ REACH Pr J q r /\
     |==((p AND J) IMP q)
     ==>
     REACH Pr J p r

REACH_SUBST:
  |- !Pr J p q r s.
     (WRITE Pr) CONF r /\ (WRITE Pr) CONF s /\
     |==((r AND J) IMP p) /\
     |==((q AND J) IMP s) /\
     REACH Pr J p q
     ==>
     REACH Pr J r s);

REACH_IMP_CONF:
  |- !Pr J p q.
     REACH Pr J p q
     ==>
     (WRITE Pr) CONF p /\ (WRITE Pr) CONF q

REACH_IMP_STABLE:
  |- !Pr J p q. REACH Pr J p q ==> STABLE Pr J

REACH_IMP_LEADSTO:
  |- !Pr J p q.
     (WRITE Pr) CONF J /\ REACH Pr J p q
     ==>
     LEADSTO Pr(p AND J)q

REACH_CANCEL:
  |- !Pr J p q r s.
     (WRITE Pr) CONF q /\
     REACH Pr J p(q OR r) /\
     REACH Pr J r s
     ==>
     REACH Pr J p(q OR s)

REACH_GEN_PSP:
  |- !Pr J L p q a b.
     (WRITE Pr) CONF a /\ (WRITE Pr) CONF b /\
     REACH Pr J p q /\
     UNLESS Pr(a AND L)b /\ STABLE Pr L
     ==>
     REACH Pr(J AND L)(p AND a)((q AND a) OR b)

REACH_STAB_MONO:
  |- !Pr J H p q.
     STABLE Pr H /\ REACH Pr J p q
     ==>
     REACH Pr(H AND J)p q

REACH_STABLE_SHIFT:
  |- !Pr J a p q.
     (WRITE Pr) CONF a /\
     REACH Pr(J AND a)p q /\ STABLE Pr J
     ==>
     REACH Pr J(p AND a)q

REACH_WF_INDUCT:
  |- !U :: ADMIT_WF_INDUCTION. !Pr M J p q.
     (WRITE Pr) CONF q /\
     (!m. REACH Pr J (p AND (\s. M s = m))
                     ((p AND (\s. U(M s)m)) OR q))
     ==>
     REACH Pr J p q

REACH_EVENTUALLY_FALSE:
  |- !U :: ADMIT_WF_INDUCTION. !Pr M J p.
     (WRITE Pr) CONF p /\
    (!m. (WRITE Pr) CONF (\s. U(M s)m)) /\
    (!m. REACH Pr J((NOT p) AND (\s. M s = m))
                 (\s. U(M s)m))
    ==>
    REACH Pr J TT p

REACH_COMPLETION:
  |- !W Pr J P Q b.
    CHF_FINITE W /\ UNITY Pr /\
    (WRITE Pr) CONF b /\
    STABLE Pr J /\
    (!i :: W. REACH Pr J(P i)((Q i) OR b)) /\
    (!i :: W. UNLESS Pr((Q i) AND J)b)
    ==>
    REACH Pr J (!! i :: W. P i)
               ((!! i :: W. Q i) OR b)
```

A.10 Convergence

The convergence operator is defined as follows in HOL:

```
CON:
  |- !Pr J p q.
     CON Pr J p q =
     (WRITE Pr) CONF q /\
     (?q'. REACH Pr J p(q' AND q) /\ STABLE Pr(q' AND (q AND J)))
```

Below is a list of its basic properties. The operator was discussed in Chapter 5. Most of the properties below were also listed there.

```
CON_IMP_LIFT:
  |- !Pr J p q.
    UNITY Pr /\
    (WRITE Pr) CONF p /\ (WRITE Pr) CONF q /\
    STABLE Pr J /\ STABLE Pr(q AND J) /\
    |==((p AND J) IMP q)
    ==>
    CON Pr J p q

CON_ENSURES_LIFT:
  |- !Pr J p q.
    (WRITE Pr) CONF p /\ (WRITE Pr) CONF q /\
     STABLE Pr J /\ STABLE Pr (q AND J) /\
     ENSURES Pr (p AND J) q
     ==>
     CON Pr J p q);

CON_REACH_LIFT:
  |- !Pr J p q.
    REACH Pr J p q /\ STABLE Pr(q AND J)
    ==>
    CON Pr J p q

CON_REFL:
  |- !Pr J p.
    UNITY Pr /\ (WRITE Pr) CONF p /\
    STABLE Pr J /\ STABLE Pr(p AND J)
    ==>
    CON Pr J p p

CON_gMONO:
  |- !Pr J p q r.
    (WRITE Pr) CONF r /\
    CON Pr J p q /\ |==((q AND J) IMP r)
    ==>
    CON Pr J p r

CON_gANTIMONO:
  |- !Pr J p q r.
    (WRITE Pr) CONF p /\
    CON Pr J q r /\ |==((p AND J) IMP q)
    ==>
    CON Pr J p r

CON_SUBST:
  |- !Pr J p q r s.
    (WRITE Pr) CONF r /\ (WRITE Pr) CONF s /\
    |==((r AND J) IMP p) /\
    |==((q AND J) IMP s) /\
    CON Pr J p q
    ==>
    CON Pr J r s

CON_SPIRAL:
  |- !Pr J p q r.
    CON Pr J p q /\ CON Pr J q r
    ==>
    CON Pr J p(q AND r)

CON_TRANS:
  |- !Pr J p q r.
    CON Pr J p q /\ CON Pr J q r
    ==>
    CON Pr J p r


CON_LDISJ:
  |- LDISJ(CON Pr J))

CON_DISJ:
  |- DISJ W(CON Pr J)

CON_CONJ:
  |- !W Pr J P Q.
    CHF_FINITE W /\
    UNITY Pr /\
    STABLE Pr J /\
    (!i :: W. CON Pr J(P i)(Q i))
    ==>
    CON Pr J(!! i :: W. P i)(!! i :: W. Q i)
```

```
CON_IMP_REACH:
  |- !Pr J. (CON Pr J) SUBREL (REACH Pr J)

CON_CONF_LIFT:
  |- !Pr J p q.
     CON Pr J p q
     ==>
     (WRITE Pr) CONF p /\ (WRITE Pr) CONF q

CON_IMP_STABLE:
  |- !Pr J p q. CON Pr J p q ==> STABLE Pr J

CON_STABLE_SHIFT:
  |- !Pr J a p q.
     (WRITE Pr) CONF a /\
     CON Pr (J AND a) p q /\ STABLE Pr J
     ==>
     CON Pr J (p AND a) q

CON_WF_INDUCT: % bounded progress induction
                 principle for convergence %
  |- ADMIT_WF_INDUCTION LESS /\
     CON Pr J q q /\
     (!m. CON Pr J
              (p AND (\s. M s = m))
              ((p AND (\s. LESS(M s)m)) OR q))
     ==>
     CON Pr J p q

CON_BY_sWF_i: % the round decomposition
                principle %
  |- sWF A U /\ ~(A = {}) /\ FINITE A /\
     STABLE Pr J /\
     (!y::\y. y IN A.
         CON Pr
             (J AND
              (!! x :: \x. x IN A /\ U x y. Q x))
             TT
             (Q y))
     ==>
     CON Pr J TT(!! y :: \y. y IN A. Q y)
```

A.11 Parallel Composition

Below is the definition of UNITY parallel composition together with some other operators used later. A list of theorems stating the compositionality of various UNITY operators follows. This includes the Singh law for the reach operator, 'until'-like compositionality, compositionality achieved through fix point, and the transparency laws.

```
PAR: % parallel composition of programs %
  |- !Pr Qr.
     Pr PAR Qr =
       ( PROG Pr) OR (PROG Qr),
         (INIT Pr) AND (INIT Qr),
         (READ Pr) OR (READ Qr),
         (WRITE Pr) OR (WRITE Qr)

DVa: % 'shared' variables %
  |- !Pr Qr.
     DVa Pr Qr = (READ Pr) AND (WRITE Qr)

UP_DEF: % lift a relation to the predicate
          level %
  |- !V R C.
     UP V R C = (\s. !x. V x ==> R(s x)(C x))

WD_DEF: % write disjoint programs %
  |- !Pr Qr.
     Pr <w> Qr = (!v. WRITE Pr v ==> ~WRITE Qr v)
```

```
PAR_UNIT:
  |- !Pr. Pr PAR (FF,TT,FF,FF) = Pr

PAR_SYM:
  |- !Pr Qr. Pr PAR Qr = Qr PAR Pr

PAR_ASSOC:
  |- !Pr Qr Rr.
     (Pr PAR Qr) PAR Rr = Pr PAR (Qr PAR Rr))

UNITY_PAR:
  |- !Pr Qr.
     UNITY Pr /\ UNITY Qr ==> UNITY(Pr PAR Qr)

CONF_PAR:
  |- !Pr Qr p.
     (WRITE Pr) CONF p \/ (WRITE Qr) CONF p
     ==>
     (WRITE(Pr PAR Qr)) CONF p

UNLESS_PAR_i:
  |- UNLESS Pr p q /\ UNLESS Qr p q
     =
     UNLESS(Pr PAR Qr)p q

STABLE_PAR_i:
  |- STABLE Pr p /\ STABLE Qr p
     =
     STABLE(Pr PAR Qr)p

Inv_PAR:
  |- !J Pr Qr.
     Inv Pr J /\ Inv Qr J ==> Inv(Pr PAR Qr)J

ENSURES_PAR:
  |- !Pr :: UNITY. !p q Qr.
     UNLESS Pr p q /\ ENSURES Qr p q
     ==>
     ENSURES(Pr PAR Qr)p q
```

```
REACH_SINGH_g: % Singh law for reach %
  |- UNITY Qr /\
     (WRITE(Pr PAR Qr)) CONF a /\
     (WRITE(Pr PAR Qr)) CONF b /\
     ((WRITE Pr) OR ((READ Pr)
               AND (WRITE Qr))) CONF p0 /\
     STABLE(Pr PAR Qr)J /\
     (!C. UNLESS Qr (J AND (a AND (UP(DVa Pr Qr)$= C)))
                 b ) /\
     REACH Pr (J AND p0) p q
     ==>
     REACH (Pr PAR Qr)
           J
           (p AND (p0 AND a))
           (q OR ((NOT a) OR ((NOT p0) OR b)))

UNTIL_COMPO1:
  |- UNITY Qr /\
     STABLE Qr J /\
     (!C. UNLESS Qr
              (J AND (p AND (UP(DVa Pr Qr)$= C)))
              q ) /\
     UNLESS Pr( p AND J) q /\
     REACH Pr J p q
     ==>
     REACH (Pr PAR Qr) J p q

UNTIL_COMPO2',
  |- UNITY Qr /\
     ((WRITE Pr) OR ((READ Pr)
               AND (WRITE Qr))) CONF p /\
     STABLE (Pr PAR Qr) J /\
     (!C. UNLESS Qr
              (J AND (p AND (UP(DVa Pr Qr)$= C)))
              q ) /\
     REACH Pr (J AND p) TT q
     ==>
     REACH (Pr PAR Qr) J p q

REACH_COMPO_BY_FPp:
  |- UNITY Qr /\ FPp Qr a /\ STABLE Pr(J AND a) /\
     REACH Pr J p q
     ==>
     REACH(Pr PAR Qr)(J AND a)p q

REACH_TRANSPARANT: % transparency law for reach %
  |- !Pr Qr J p q.
      UNITY Qr /\ Pr <w> Qr /\
      STABLE Qr J /\ REACH Pr J p q
      ==>
      REACH(Pr PAR Qr)J p q

CON_TRANSPARANT:
  % transparency law for convergence %
  |- !Pr Qr J p q.
     UNITY Qr /\ Pr <w> Qr /\
     STABLE Qr J /\ CON Pr J p q
     ==>
     CON(Pr PAR Qr)J p q

REACH_CONJ_WD:
  |- !Pr Qr J p q r s.
     Pr <w> Qr /\
     REACH Pr J p q /\ STABLE Pr(q AND J) /\
     REACH Qr J r s /\ STABLE Qr(s AND J)
     ==>
     REACH(Pr PAR Qr)J(p AND r)(q AND s)

CON_CONJ_WD:
  |- !Pr Qr J p q r s.
     Pr <w> Qr /\ CON Pr J p q /\ CON Qr J r s
     ==>
     CON(Pr PAR Qr)J(p AND r)(q AND s)

REACH_SPIRAL:
  |- !Pr Qr J p q.
     Pr <w> Qr /\
     STABLE Pr(J AND q) /\ STABLE Qr J /\
     REACH Pr J p q /\ REACH Qr(J AND q)TT r
     ==>
     REACH(Pr PAR Qr)J p(q AND r)
```

A.12 Lattice Theory

Below are some definitions from lattice theory which we will need later.

```
PO: % partial order %
  |- !r. PO r = REFL r /\ TRANS r /\ ANTI_SYM r

IS_LB: % lower bound %
  |- !r X a. IS_LB r X a = (!b. b IN X ==> r a b)

isCAP:
  |- !r X c.
     isCAP r X c
     =
     IS_LB r X c /\ (!b. IS_LB r X b ==> r b c)

CAP: % greatest lower bound %
  |- !r X. (?a. isCAP r X a) ==> isCAP r X (CAP r X)

CAP_PLa: % (complete) CAP-lattice %
  |- !r.
     CAP_PLa r = PO r /\ (!X. ?b. isCAP r X b)

Bot: % bottom %
  |- !r. Bot r = CAP r UNIV

Top_ADEF: % top %
  |- CAP_PLa r ==> (Top r = CAP r{})

IMAGE_DEF : % mapping on sets %
  |- !f s. *> f s = {f x | x IN s}

CAP_Junct: % CAP-junctivity of a function %
  |- !r s f.
     CAP_Junct r s f =
     (!X. ~(X = {}) ==>
          (f(CAP r X) = CAP s(*> f X)))
```

```
CAP_Distr: % CAP-distributivity of a function %
  |- !r s f.
```

```
CAP_Distr r s f =
(!X. f(CAP r X) = CAP s(*> f X))
```

A.13 Minimal-Distance-Like Functions

Below is the definition of minimal-distance-like (minimal-cost-like) functions, along with some other definitions required later. A list of properties of such a function follows.

```
ONE_ONE_DEF: % define injective functions %
  |- !f. ONE_ONE f =
        (!x1 x2. (f x1 = f x2) ==> (x1 = x2))

GRAPH: % define a graph (network) %
  |- !V N. GRAPH(V,N) =
          (!a. a IN V ==> (N a) SUBSET V)

PATH: % define what is a path in a network %
  |- (!N A b. PATH N A b[] = b IN A) /\
     (!N A b a s. PATH N A b(CONS a s)
                  =
                  a IN (N b) /\ PATH N A a s)

PATHS:
  |- !N A b. PATHS N A b = {s | PATH N A b s}

PathCost: % define the cost of a path %
  |- (!addW e b. PathCost addW e b[] = e) /\
     (!addW e b a s.
         PathCost addW e b(CONS a s)
         =
         addW a b(PathCost addW e a s))

MiCost: % define minimal cost from source to
        destination %
  |- !N r addW e A b.
     MiCost N r addW e A b =
     CAP r (*> (PathCost addW e b) (PATHS N A b))

FishBone:
  |- !addW b c r a.
     >+- addW b c(r,a) = addW b c r,b

FishTail:
  |- !r1 r2 addW b c x a.
     >++ r1 r2 addW b c(x,a)
     =
     ((x = Top r1) => Top(r1 >>> r2)
                    | >+- addW b c(x,a))

CrabPlus:
  |- !addW a b X.
     >+< addW a b X = addW a b(FST X),SND X)

Crab:
  |- !r1 r2 addW b c x a.
     >-< r1 r2 addW b c(x,a) =
     ((x = Top r1) => Top(r1 >>> r2)
                    | >+< addW b c(x,a))
```

```
MiCost_SELF:
  |- CAP_PLa r /\ (e = Bot r) /\ a IN A
     ==>
    (MiCost N r addW e A a = e)

MiCost_SPLIT1:
  |- CAP_PLa r /\
     (!a b. CAP_Distr r r (addW a b)) /\
     ~c IN A
     ==>
     (MiCost N r addW e A c
        =
        CAP r (*> (\b. addW b c (MiCost N r addW e A b)) (N c)))

MiCost_LESS_NEIGHBOR1:
  |- CAP_PLa r /\
     (!a b. CAP_Distr r r (addW a b)) /\
     a IN (N b)
     ==>
     r (MiCost N r addW (Bot r)A b)
       (addW a b (MiCost N r addW (Bot r)A a))

MiCost_IN_NEIGHBOR1:
  |- CAP_PLa r /\ CAP_Closed r /\
     (!a b. CAP_Distr r r (addW a b)) /\
     ~(N b = {}) /\ A SUBSET V /\
     b IN V /\ ~b IN A
     ==>
     (MiCost N r addW e A b) IN
     (*> (\a. addW a b (MiCost N r addW e A a)) (N b))

Fish_Distr:
  |- CAP_PLa r1 /\ CAP_PLa r2 /\
     CAP_Closed r1 /\ (!a b. CAP_Distr r1 r1 (addW a b))
     ==>
     CAP_Distr (r1 >>> r2) (r1 >>> r2) (>++ r1 r2 addW a b)

FST_MiCost_Fish:
  |- CAP_PLa r1 /\ CAP_PLa r2 /\
     (!a b. CAP_Distr r1 r1 (addW a b))
     ==>
     (FST o (MiCost N (r1 >>> r2) (>++ r1 r2 addW) (e,ep) a)
        =
        MiCost N r1 addW e a)

SND_MiCost_Fish:
  |- GRAPH(V,N) /\ CAP_PLa r1 /\ CAP_PLa r2 /\
     CAP_Closed r1 /\ CAP_Closed r2 /\
     (!a b. CAP_Distr r1 r1(addW a b)) /\
     ~(MiCost N r1 addW e A b = Top r1) /\
     ~(N b = {}) /\ A SUBSET V /\ ~(A = {}) /\ b IN V /\ ~b IN A
     ==>
     (SND (MiCost N (r1 >>> r2) (>++ r1 r2 addW)(e,ep)A b))
          IN (N b)

MiCost_Fish_THM:
  |- GRAPH(V,N) /\ CAP_PLa r1 /\ CAP_PLa r2 /\
     CAP_Closed r1 /\ CAP_Closed r2 /\
     (!a b. CAP_Distr r1 r1 (addW a b)) /\
     ~(MiCost N r1 addW e A b = Top r1) /\
     ~(N b = {}) /\ A SUBSET V /\ ~(A = {}) /\ b IN V /\ ~b IN A /\
     (b0 = SND (MiCost N (r1 >>> r2) (>++ r1 r2 addW) (e,ep) A b))
      ==>
     (MiCost N r1 addW e A b = addW b0 b(MiCost N r1 addW e A b0))


Crab_Distr:
  |- CAP_PLa r1 /\ CAP_PLa r2 /\ CAP_Closed r1 /\
     (!a b. ONE_ONE (addW a b)) /\
     (!a b. CAP_Distr r1 r1(addW a b))
     ==>
     CAP_Distr (r1 >>> r2) (r1 >>> r2) (>-< r1 r2 addW a b)

FST_MiCost_Crab:
  |- CAP_PLa r1 /\ CAP_PLa r2 /\ (!a b. CAP_Distr r1 r1(addW a b))
     ==>
    (FST (MiCost N (r1 >>> r2) (>-< r1 r2 addW) (e,X)A a)
          =
          MiCost N r1 addW e A a)

SND_MiCost_Crab:
  |- CAP_PLa r1 /\ CAP_PLa r2 /\ CAP_Closed r1 /\
     (!a b X. r1 X (addW a b X)) /\
     (!a b. CAP_Distr r1 r1 (addW a b)) /\
     ~(MiCost N r1 addW e A b = Top r1) /\
     b IN V /\ A SUBSET V /\ ~(A = {})
     ==>
     (SND (MiCost N (r1 >>> r2) (>-< r1 r2 addW) (e,X)A b)
           = X)
```

A.14 Round Solvability

Below are the round solvability theorems for minimal-distance-like functions, for minimal-distance-like functions with best neighbors recording, and for broadcast functions. First some definitions:

```
OKmc: % define the OK predicate for minimum-distance-like functions %
  |- !r n f a X.
      OKmc r n f a X =
         (r (f a) n ==> (X = f a))  /\  (r n(f a) ==> r n X)

GENmc: % define the generator for minimum-distance-like functions %
  |- !N r addW e f A b.
      GENmc N r addW e f A b =
           (b IN A => e | CAP r (*> (\a. addW a b (f a)) (N b)))

OKbc: % define the OK predicate for a broadcast function %
  |- !P r n f a X.
      OKbc P r n f a X =
      OKmc r n f a (FST X) /\ (I_r r(f a)n ==> P(SND X))

GENbc: % define generator for broadcasting %
  |- !N r1 r2 addW e f g A b.
      GENbc N r1 r2 addW e f g A b =
      (b IN A => (e,g b) |
                  CAP (r1 >>> r2) (*> (\a. >-< r1 r2 addW a b (f a)) (N b)))
```

```
MiCost_Round_Solv: % the round solvability of a minimum-distance-like
                     function %
  |- GRAPH(V,N) /\
     CAP_PLa r /\ CAP_Closed r /\
     (!a b. CAP_Distr r r(addW a b)) /\
     ~(MiCost N r addW e A a = Top r) /\ (e = Bot r) /\
     (!n a b. ~(n = Top r) ==> I_r r n(addW a b n)) /\
     A SUBSET V /\ ~(A = {}) /\ a IN V /\
     (!a0 m.
        a0 IN (N a) /\ I_r r m n
        ==>
        OKmc r m (MiCost N r addW e A) a0 (f a0))
     ==>
     OKmc r n (MiCost N r addW e A) a (GENmc N r addW e f A a)

MiCostFish_Round_Solv:
   % the round solvabilit minimum-distance-like functions,
     with best neighbors recording %
   |- GRAPH(V,N) /\ CAP_PLa r1 /\ CAP_PLa r2 /\
      CAP_Closed r1 /\ CAP_Closed r2 /\
      (!a b. CAP_Distr r1 r1(addW a b)) /\
      ~(MiCost N r1 addW e A a = Top r1) /\
      (e = Bot r1) /\ (ep = Bot r2) /\
      (r3 = r1 >>> r2) /\
      (!a b. CAP_Distr r1 r1(addW a b)) /\
      (!n a b. ~(n = Top r1) ==> I_r r1 n(addW a b n)) /\
      A SUBSET V /\ ~(A = {}) /\ a IN V /\
      (!a0 m.
           a0 IN (N a) /\ I_r r3 m n
           ==>
           OKmc r3 m (MiCost N r3 (>++ r1 r2 addW) (e,ep)A) a0 (f a0))
      ==>
      OKmc r3 n
           (MiCost N r3(>++ r1 r2 addW)(e,ep)A)
```

```
         a
         (GENmc N r3(>++ r1 r2 addW)(e,ep)f A a)

BC_Round_Solv: % the round solvability of broadcast functions %
  |- GRAPH(V,N) /\ CAP_PLa r1 /\ CAP_PLa r2 /\
     CAP_Closed r1 /\ CAP_Closed r2 /\
     (e = Bot r1) /\
     (!a b. CAP_Distr r1 r1(addW a b)) /\
     (!n a b. ~(n = Top r1) ==> I_r r1 n(addW a b n)) /\
     (!a. a IN A ==> P(g a)) /\
     ~(MiCost N r1 addW e A a = Top r1) /\
     A SUBSET V /\ ~(A = {}) /\ a IN V /\
     (!a0 m.
         a0 IN (N a) /\ I_r r1 m n ==>
         OKbc P r1 m (MiCost N r1 addW e A) a0 (f a0))
     ==>
     OKbc P r1 n (MiCost N r1 addW e A) a (GENbc N r1 r2 addW e f g A a)
```

A.15 The FSA Algorithm

Below is the definitions of the FSA and the broadcast algorithms, followed by several theorems stating their self-stabilizing properties. However, first let us say something about the notation being used. The function CHF used in the definition of the algorithms is a function to convert sets to predicates:

```
|- !x s. x IN s = CHF s x
```

That is, if s is a set, then CHF s is a predicate characterizing s. Using predicates instead of sets often saves some proof steps. Another function which needs some explanation is GSPEC. It is used to encode set abstractions in HOL:

```
|-  GSPEC  (\x. (f x, g x)) = {f x | g x}
```

The notation {...} is just the result of pretty printing in HOL, and that is also how we usually denote sets. However, in a formula like $\{f.x.y|x \in A\}$ it is not clear whether the 'y' is a constant or a bound variable like x. We usually decide this from the context. HOL cannot do this, so in a dubious case such as above we have to revert to the GSPEC notation.

```
ONE_ONE2_DEF: % define injective functions %
  |- !g. ONE_ONE2 g =
        (!a b c d. (g a b = g c d)
                   ==>
                  (a = c) /\ (b = d))

Proper_DEF:
  |- !P V.
     Proper P V = (!a. a IN V ==> UNITY(P a))

Distinct_DEF:
  |- !f g. Distinct f g = (!a b c. ~(g b c = f a))

dClosed:
  |- !r A. dClosed r A =
         (!m n. n IN A /\ r m n ==> m IN A)
```

```
Resolve_DEF: % the round solvability condition %
  |- !N A R Res ok n a.
     Resolve N (A,R) Res ok n a =
       (!f. (!m b. m IN A /\ R m n /\ b IN (N a)
                ==> ok m b (f b))
             ==>
             ok n a (Res a f))

MSP: % lift a predicate to the state-
       predicate level %
  |- !ok n a x s. MSP ok n a x s = ok n a(s(x a))

Comp_DEF: % an action 'x.a := Res.a.(r.a)' %
  |- !Res a x r s t.
```

```
      Comp Res a x r s t
      =
      (!v. (v = x a) ==> (t v = Res a(s o (r a))))
      /\
      (!v. ~(v = x a) ==> (t v = s v))

Copy_DEF: % a copy action 'r.b.a := x.a' %
  |- !a b x r s t.
      Copy a b x r s t =
      (!v. (v = r b a) ==> (t v = (s o x)a)) /\
      (!v. ~(v = r b a) ==> (t v = s v))

FSA : % the FSA algorithm %
  |- !Res V N x r a.
      FSA Res(V,N)x r a =
      ( CHF({Comp Res a x r} |_|
           (GSPEC (\b. (Copy a b x r,
                        (b IN V /\ a IN (N b)))))),
        TT,
        CHF({x a} |_|
            ({r a b | b IN V} |_|
             {r b a | b IN V})),
        CHF({x a} |_| {r b a | b IN V}) )

Comp_bc_DEF: % the core action of the
               broadcast program %
  |- !N R1 R2 addW e g Org a x r s t.
     Comp_bc N R1 R2 addW e g Org a x r s t
     =
     (!v. (v = x a)
          ==>
          (t v = GENbc N R1 R2 addW e
                       (s o (r a))
                       (SND o (s o g)) Org a))
    /\
    (!v. ~(v = x a) ==> (t v = s v))

BC: % the broadcast program %
  |- !V N R1 R2 addW e g Org x r a.
      BC (V,N) R1 R2 addW e g Org x r a =
      ( CHF ({Comp_bc N R1 R2 addW e g Org a x r}
               |_|
               (GSPEC (\b. (Copy a b x r,
                            (b IN V /\ a IN (N b)))))),
        TT,
        CHF({x a,g a} |_|
           ({r a b | b IN V} |_|
            {r b a | b IN V})),
      CHF({x a} |_| {r b a | b IN V}))

Net_DEF:
  |- !P V. Net P V = gPAR (*> P V)
  % NOTE: gPAR P V denotes (PAR i: i IN V: P i) %

dataOK:
  |- !V OK n x.
     dataOK V OK n x
     =
     (\s. !a. a IN V ==> OK n a x s)

comOK:
  |- !V N OK n r.
     comOK(V,N)OK n r =
       (\s. !a b. a IN V /\ b IN (N a)
                  ==>
                  OK n b (r a) s)

MDC_DEF: % the specification of the FSA
           algorithm %
  |- !Pr J V A OK x.
      MDC Pr J V A OK x
      =
      CON Pr J TT (!! n :: CHF A. dataOK V OK n x)

MDCC_DEF: % a stronger variant of MDC %
  |- !Pr J V N A OK x r.
      MDCC Pr J (V,N) A OK x r
      =
      CON Pr J TT (!! n :: CHF A. (dataOK V OK n x)
                                  AND (comOK(V,N)OK n r))
```

```
FSA_sat_MDC: % the general theorem for the FSA algorithm %
  |- sWF A R /\ ~(A = {}) /\ ~(V = {}) /\
     FINITE A /\ FINITE V /\ GRAPH(V,N) /\
     (!n a. n IN A /\ a IN V ==> Resolve N (A,R) Res ok n a) /\
     Proper(FSA Res(V,N)x r)V /\
     ONE_ONE x /\ ONE_ONE2 r /\ Distinct x r
     ==>
     MDC (Net (FSA Res (V,N) x r) V) TT V A (MSP ok) x

MiCost_sat_MDC:
  % the theorem for the FSA algorithm, instantiated to compute
     minimal-distance-like functions %
  |- ~(A = {}) /\ ~(V = {}) /\ FINITE A /\ FINITE V /\ GRAPH(V,N) /\
     CAP_PLa R /\ CAP_Closed R /\
     (e = Bot R) /\ (!a b. CAP_Distr R R (addW a b)) /\
     (!a. a IN V ==> ~(MiCost N R addW e Org a = Top R)) /\
     (!n a b. ~(n = Top R) ==> I_r R n(addW a b n)) /\
     Org SUBSET V /\ ~(Org = {}) /\ dClosed R A /\
     (Res = (\a f. GENmc N R addW e f Org a)) /\
     (ok = (\n a val. OKmc R n(MiCost N R addW e Org)a val)) /\
     Proper(FSA Res(V,N)x r)V /\ ONE_ONE x /\ ONE_ONE2 r /\ Distinct x r
     ==>
     MDC (Net (FSA Res (V,N) x r) V) TT V A (MSP ok) x
```

```
MiCostFish_sat_MDC:
  % the theorem for the FSA algorithm, instantiated to compute
    minimal-distance-like functions with best-neighbors
    recording %
  |- ~(A = {}) /\ ~(V = {}) /\ FINITE A /\ FINITE V /\ GRAPH(V,N) /\
     CAP_PLa R1 /\ CAP_PLa R2 /\ CAP_Closed R1 /\ CAP_Closed R2 /\
     (!a b. CAP_Distr R1 R1 (addW a b)) /\
     (!a. a IN V ==> ~(MiCost N R1 addW e Org a = Top R1)) /\
     (!n a b. ~(n = Top R1) ==> I_r R1 n (addW a b n)) /\
     Org SUBSET V /\ ~(Org = {}) /\ dClosed(R1 >>> R2)A /\
     (e = Bot R1) /\ (ep = Bot R2) /\
     (Res = (\a f. GENmc N(R1 >>> R2)(>++ R1 R2 addW)(e,ep)f Org a)) /\
     (ok =  (\n a val. OKmc (R1 >>> R2) n
                          (MiCost N(R1 >>> R2)(>++ R1 R2 addW)(e,ep)Org)
                          a val)) /\
     Proper (FSA Res(V,N)x r) V /\
     ONE_ONE x /\ ONE_ONE2 r /\ Distinct x r
     ==>
     MDC (Net (FSA Res (V,N) x r) V) TT V A (MSP ok) x

BC_sat_MDC: % the theorem for self-stabilizing broadcast %
  |- ~(A = {}) /\ ~(V = {}) /\ FINITE A /\ FINITE V /\ GRAPH(V,N) /\
     CAP_PLa r1 /\ CAP_PLa r2 /\ CAP_Closed r1 /\ CAP_Closed r2 /\
     (!a b. CAP_Distr r1 r1 (addW a b)) /\
     (!a. a IN V ==> ~(MiCost N r1 addW e Org a = Top r1)) /\
     (!n a b. ~(n = Top r1) ==> I_r r1 n (addW a b n)) /\
     dClosed r1 A /\ ONE_ONE x /\ Distinct x r /\ ONE_ONE2 r /\
     Org SUBSET V /\ ~(Org = {}) /\
     (!s a. J s /\ a IN Org ==> (P o (SND o (s o g)))a) /\
     (e = Bot r1) /\
     (ok = (\n a val. OKbc P r1 n (MiCost N r1 addW e Org) a val)) /\
     (Pr = BC (V,N) r1 r2 addW e g Org x r) /\
     (!a. a IN V ==> STABLE (Pr a) J) /\
     Proper Pr V
      ==>
     MDC (Net Pr V) J V A (MSP ok) x

BC_sat_MDC2: % a stronger variant of BC_SAT_MDC %
  |- ~(A = {}) /\ ~(V = {}) /\ FINITE A /\ FINITE V /\ GRAPH(V,N) /\
     CAP_PLa r1 /\ CAP_PLa r2 /\ CAP_Closed r1 /\ CAP_Closed r2 /\
     (!a b. CAP_Distr r1 r1 (addW a b)) /\
     (!a. a IN V ==> ~(MiCost N r1 addW e Org a = Top r1)) /\
     (!n a b. ~(n = Top r1) ==> I_r r1 n(addW a b n)) /\
     dClosed r1 A /\ ONE_ONE x /\ Distinct x r /\ ONE_ONE2 r /\
     Org SUBSET V /\ ~(Org = {}) /\
     (!s a. J s /\ a IN Org ==> (P o (SND o (s o g)))a) /\
     (e = Bot r1) /\
     (ok = (\n a val. OKbc P r1 n (MiCost N r1 addW e Org)a val)) /\
     (Pr = BC(V,N)r1 r2 addW e g Org x r) /\
     (!a. a IN V ==> STABLE(Pr a)J) /\
     Proper Pr V
     ==>
     MDCC (Net Pr V) J (V,N) A (MSP ok) x r
```

A.16 The Domain Level FSA Algorithm

Below are the definition of the domain level FSA algorithm, its specification (DA0), and a theorem detailing the conditions required for the algorithm to satisfy DA0.

```
applyGen_DEF:
  |- !gen d cp B b s t.
     applyGen gen d cp B b s t =
       (!v. (v = d B b) ==> ((SND o t) v = gen (SND o s o (cp B b)))) /\
       (!v. (v = d B b) ==> ((FST o t) v = (FST o s) v)) /\
       (!v. ~(v= d B b) ==> (t v = s v))

gFSA_DEF:
  |- !V gen d cp B b.
     gFSA V gen d cp B b =
     (CHF{applyGen gen d cp B b}, TT, CHF((*> (cp B b) V) |_| {d B b}), CHF{d B b})

Boder_DEF:
  |- !Nn Nd C b. Border Nn Nd C B = {c | c IN (Nd C) /\ (?b. b IN (Nd B) /\  b IN (Nn c))}

Sel_DEF:
  |- !Nn Nd C B c.
      c IN (Border Nn Nd C B) ==> (Sel Nn Nd B c) IN (Nd B) /\ (Sel Nn Nd B c) IN (Nn c)

ccFSA_DEF:
  |- !Nd Nn r1 r2 addW_bc e_bc d cp r B C c.
     ccFSA Nd Nn r1 r2 addW_bc e_bc d cp r  B C c =
        BC (Nd C, (\c1. {c2 | c2 IN (Nd C) /\ c2 IN (Nn c1)}))
            r1 r2 addW_bc e_bc
           ((d B) o (Sel Nn Nd B))
           (Border Nn Nd C B)
           (\c'. cp C c' B)
           (r C B)
           c

cFSA_ADEF:
  |- !Nd Nn r1 r2 addW_bc e_bc d cp r B C.
     cFSA Nd Nn r1 r2 addW_bc e_bc d cp r B C =
     Net (ccFSA Nd Nn r1 r2 addW_bc e_bc d cp r  B C) (Nd C)

dFSA_Collect:
  |- !Nd Nn V N r1 r2 addW_bc e_bc Gen d cp r.
     dFSA_Collect Nd Nn (V,N) r1 r2 addW_bc e_bc Gen d cp r =
       (GSPEC(\(B,b). (gFSA V (Gen (V,N) B) d cp  B b, (B IN V /\ b IN (Nd B)))))
        |_|
       (GSPEC(\(B,C). (cFSA Nd Nn r1 r2 addW_bc e_bc d cp r B C, (C IN V /\ B IN (N C))))

flatUprogs:
   |- !Pset. flatUprogs Pset =
       ((??P::(\P. P IN Pset). PROG P), (!!P::(\P. P IN Pset). INIT P),
        (??P::(\P. P IN Pset). READ P), ??P::(\P. P IN Pset). WRITE P))

dFSA_DEF:
  |- !Nd Nn V N r1 r2 addW_bc e_bc Gen d cp r.
     dFSA Nd Nn (V,N) r1 r2 addW_bc e_bc Gen d cp r =
     flatUprogs (dFSA_Collect Nd Nn (V,N) r1 r2 addW_bc e_bc Gen d cp r)

D_dataOK:
  |- !V Nd ok d n.
       D_dataOK V Nd ok d n =
       (\s. !b B. b IN (Nd B) /\ B IN V ==> ok n B ((SND o s) (d B b)))

D_comOK:
  |- !V N ccok n. D_comOK V N ccok n =
     (\s. !B C. C IN V /\ B IN (N C) ==>  ccok n B C s)

DAO_DEF:
  |- !P J Nd (V,N) A ok ccok d r0.
     DAO P J Nd (V,N) A ok ccok d r0 =
     CON  P
          J
          TT
         (!!n::(\n. n IN A). (D_dataOK V Nd ok d n) AND (D_comOK V N ccok n))
```

```
ccOK:
  |- !Nd Nn r1 e1 addWbc ok cp r n B C.
     ccOK Nd Nn r1 e1 addWbc ok cp r n B C =
     (!!c::(\c. c IN (Nd C)).
           (\s. OKbc (ok n B) r1 (Top r1)
                    (MiCost (\c1. {c2 | c2 IN (Nd C) /\ c2 IN (Nn c1)})
                            r1 addWbc e1 (Border Nn Nd C B))
                    c
                    (s (cp C c B))))
      AND
     (!!c::(\c. c IN (Nd C)).
        (!!c'::(\c'. c' IN (Nd C) /\ c' IN (Nn c)).
              (\s. OKbc (ok n B) r1 (Top r1)
                   (MiCost (\c1. {c2 | c2 IN (Nd C) /\ c2 IN (Nn c1)})
                           r1 addWbc e1 (Border Nn Nd C B))
                    c'
                   (s (r C B c c')))))
```

Here is the main theorem on the domain level FSA algorithm:

```
D_FSA_thm0:
  |- sWF A r0 /\ TRANS r0 /\ ~(A={}) /\ FINITE A /\
     CAP_PLa r1 /\ CAP_Closed r1 /\ (Top r1) IN (A_bc) /\ dClosed r1 A_bc /\
     (e1 = Bot r1) /\ FINITE A_bc /\
     CAP_PLa r2 /\ CAP_Closed r2 /\
     GRAPH(V,N) /\ ~(V={}) /\ FINITE V /\
     (!B. B IN V ==> FINITE (Nd B)) /\ (!B. B IN V ==> ~(Nd B = {})) /\
     (!B C c. C IN V /\ B IN (N C) /\ c IN (Nd C) ==>
             ~(MiCost (\c1. {c2 | c2 IN (Nd C) /\ c2 IN (Nn c1)})
                            r1 addWbc e1 (Border Nn Nd C B) c
              =
              Top r1)) /\
     (!c c'. CAP_Distr r1 r1 (addWbc c c')) /\
     (!i c c'. ~(i = Top r1) ==> I_r r1 i (addWbc c c' i)) /\
     (!B n. B IN V /\ n IN A ==> Resolve N (A,r0) (Gen (V,N)) ok n B) /\
     (!B b. B IN V /\ b IN (Nd B) ==> UNITY (gFSA V (Gen (V,N) B) d cp  B b)) /\
     (!B C c. C IN V /\ B IN (N C) /\ c IN (Nd C) ==>
              UNITY (ccFSA Nd Nn r1 r2 addWbc e1 d cp r B C c)) /\
     (!B b. B IN V /\ b IN (Nd B) ==> STABLE (gFSA V (Gen (V,N) B) d cp  B b) J) /\
     (!B C c. C IN V /\ B IN (N C) /\ c IN (Nd C) ==>
              STABLE (ccFSA Nd Nn r1 r2 addWbc e1 d cp r B C c) J) /\
     (!B C. C IN V /\ B IN (N C) ==> ~(Border Nn Nd C B ={})) /\
     (!B1 b1 B2 C1 c1. ~(d B1 b1 = cp C1 c1 B2)) /\
     (!B1 b1 B2 C1 c1 c2. ~(d B1 b1 = r C1 B2 c1 c2)) /\
     (!B1 C1 c1 c2 B2 C2 c3. ~(r C1 B1 c1 c2 = cp C2 c3 B2)) /\
     (!B1 b1 B2 b2. ~((B1,b1)=(B2,b2)) ==> ~(d B1 b1 = d B2 b2)) /\
     (!B1 C1 c1 B2 C2 c2. ~((B1,C1,c1)=(B2,C2,c2)) ==> ~(cp C1 c1 B1 = cp C2 c2 B2)) /\
     (!B1 C1 c1 c2 B2 C2 c3 c4.
         ~((B1,C1,c1,c2)=(B2,C2,c3,c4)) ==> ~(r C1 B1 c1 c2 = r C2 B2 c3 c4))
      ==>
      DAO (dFSA Nd Nn (V,N) r1 r2 addWbc e1 Gen d cp r)
          J Nd (V,N) A
          ok (ccOK Nd Nn r1 e1 addWbc ok cp r)
          d r0
```

Bibliography

[AG90] A. Arora and M.G. Gouda. Distributed reset. In *Proceedings of the 10th Conference on Foundation of Software Technology and Theoretical Computer Science*, 1990. Also in *Lecture Notes on Computer Science* vol. 472.

[AG92] A. Arora and M.G. Gouda. Closure and convergence: A foundation for fault-tolerant computing. In *Proceedings of the 22nd International Conference on Fault-Tolerant Computing Systems*, 1992.

[And92] Flemming Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

[Aro92] A. Arora. *A foundation for fault-tolerant computing.* PhD thesis, Dept. of Comp. Science, Univ. of Texas at Austin, 1992.

[Bac90] R.J.R. Back. Refinement calculus, part ii: Parallel and reactive programs. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Lectures Notes in Computer Science 430: Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 42–66. Springer-Verlag, 1990.

[BCM$^+$90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceeding of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society Press, 1990.

[BEvG94] R.C. Backhouse, J.P.H.W. Eijnde, and A.J.M. van Gasteren. Calculating path algorithms. *Science of Computer Programming*, (22):3–19, 1994.

[BGM90] J. Burns, M. Gouda, and R. Miller. Stabilization and pseudo-stabilization. Technical report, University of Texas, TR-90-13, May 1990.

[Bou92] R. Boulton. The hol arith library. Technical report, Computer Laboratory University of Cambridge, July 1992.

[Bou94] R.J. Boulton. Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, 1994.

[Bus94] H. Busch. First-order automation for higher-order-logic theorem proving. In T.F. Melham and J. Camilleri, editors, *Lecture Notes in Computer Science 859: Higher Order Theorem Proving and Its Application*, pages 97–122. Springer-Verlag, 1994.

[BvW90] R.J.R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, (2):247–272, 1990.

[BYC88] F. Bastani, I. Yen, and I. Chen. A class of inherently fault tolerant distributed programs. *IEEE Transactions on Software Engineering*, 14(1):1432–1442, 1988.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent system using temporal logic specifications. *ACM Trans. on Prog. Lang. and Sys.*, 2:224–263, Jan. 1986.

[Cho93] C.T. Chou. Predicates, temporal logic, and simulations. In *Proc. of HOL User's Group Workshop*. Dept. of Comp. Science, Univ. of British Columbia, 1993.

[CKW$^+$91] I. Chakravarty, M. Kleyn, T.Y.C. Woo, R. Bagrodia, and V. Austel. Unity to uc: A case study in the derivation of parallel programs. In J.P. Benâtre and D. le Métayer, editors, *Lecture Notes in Computer Science 574: Research Direction in High Level Prallel Programming Languages*, pages 7–20. Springer-Verlag, 1991.

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

[Con86] R.L. et al. Constable. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[Coo72] D.C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence γ*, chapter 5, pages 91–99. Edinburgh University Press, 1972.

[CYH91] N.S. Chen, H.P. Yu, and S.T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39(3):147–151, 1991.

[dBKPJ93] F.S. de Boer, J.N. Kok, C. Palamidessi, and Rutten J.J.M.M. On blocks: locality and asynchronous communication. In J.W. de Bakker and W.P. de Roever, editors, *Lecture Notes in Computer Science 666: Proceedings of the REX School/Workshop Semantics: Foundations and Applications*, pages 73–90, 1993.

[dBvH94] F.S. de Boer and M. van Hulst. A proof system for asynchronously communicating deterministic processes, 1994. Submitted to a conference.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, 1974.

[Dij76] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

[DIM90] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, pages 119–132, August 1990.

[GH91] M.G. Gouda and T. Herman. Addaptive programming. *IEEE Trans. Software Eng.*, 17(9), September 1991.

[GM93] Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[Goo85] D.I. Good. Mechanical proofs about computer programs. In C.A.R. Hoare and J.C. Sheperdson, editors, *Mathematical Logic and Programming Languages*, pages 55–75. Prentice-Hall, 1985.

[Gou87] M.G. Gouda. The stabilizing philosopher: Asymmetry by memory and by action. Technical Report TR-87-12, Dept. of Comp. Sciences, Univ. of Texas at Austin, 1987.

[Gri81] D. Gries. *The science of computer programming.* Springer-Verlag, 1981.

[Gri90] D. Gries. Formalism in teaching programming. In E.W. Dijkstra, editor, *Formal Development of Programs and Proofs*, pages 229–236. Addison-Wesley, 1990.

[Gru91] J. Grundy. , and it will save a lot of works —certainly for a novice— if we have a full support from hol. , and it will save a lot of works —certainly for a novice— if we have a full support from hol. , and it will save a lot of works —certainly for a novice— if we have a full support frowindow inference in the hol system. In *Proceedings of the ACM & IEEE International Tutorial and Workshop on the HOL Theorem Proving System and its Applications.* IEEE Computer Society Press, 1991.

[Her91] Ted Herman. *Adaptivity through Distributed Convergence.* PhD thesis, University of Texas at Austin, 1991.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computers programs. *Commun. Ass. Comput. Mach.*, 12:576–583, 1969.

[JS93] J. J. Joyce and C. Seger. The hol-voss system: Model-checking inside a general-purpose theorem prover. In J. J. Joyce and C. Seger, editors, *Lecture Notes in Computer Science, 780 : Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG'93,*. Springer-Verlag, 1993.

[KB87] J.E. King and W.J. Brophy. Computer entomology. *Scientific Honeyweller*, 1986-87.

[KKS91] R. Kumar, T. Kropf, and K. Schneider. Integrating a first order automatic prover in the hol environment. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications*. IEEE Computer Society Press, August 1991.

[Lam90] L. Lamport. A temportal logic of actions. Technical Report 57, Digital Systems Research Center, April 1990.

[Lån94] T. Långbacka. A hol formalization of the temporal logic of actions. In T.F. Melham and J. Camilleri, editors, *Lecture Notes in Computer Science 859: Higher Order Theorem Proving and Its Application*, pages 332–345. Springer-Verlag, 1994.

[Len93] P.J.A. Lentfert. *Distributed Hierarchical Algorithms*. PhD thesis, Utrecht University, April 1993.

[lL77] G. le Lann. Distributed systems —towards a formal approach. In B. Gilchrist, editor, *nformation Processing '77*, pages 155–160. North-Holland, 1977.

[LS93] P.J.A. Lentfert and S.D. Swierstra. Towards the formal design of self-stabilizing distributed algorithms. In P. Enjalbert, A. Finkel, and K.W. Wagner, editors, *STACS 93, Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 440–451. Springer-Verlag, February 1993.

[Pac92] J. Pachl. A simple proof of a completeness result for *leads-to* in the UNITY logic. *Information Processing Letters*, 41:35–38, 1992.

[Piz91] A. Pizzarello. An indrustial experience in the use of unity. In J.P. Benâtre and D. le Métayer, editors, *Lecture Notes in Computer Science 574: Research Direction in High Level Prallel Programming Languages*, pages 38–49. Springer-Verlag, 1991.

[Piz92] A. Pizzarello. Formal methods in corrective software maintenance. In *proceeding Formal Methods for Software Development: International Seminar*, Milano, 1992. Associazione Italiana Calcolo Automatico (AICA).

[PJ91] P.K. Pandya and M. Joseph. P-a logic –a compositional proof system for distributed programs. *Distributed Computing*, (5):37–54, 1991.

[Pra93] I.S.W.B. Prasetya. On the style of mechanical proving. In J.J. Joyce and C.J.H. Seger, editors, *LNCS 780: Higher Order Logic Theorem Proving and Its Applications*, pages 475–488. Springer-Verlag, 1993.

[Pra94] I.S.W.B. Prasetya. Error in the unity substitution rule for subscripted operators. *Formal Aspects of Computing*, 6:466–470, 1994.

[R.95] Udink. R. *Program Refinement in UNITY-like Environments.* PhD thesis, Utrecht University, 1995.

[San91] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[Sch86] D.A. Schmidt. *Denotational Semantics, a Methodology for Language Development.* Allyn and Bacon, 1986.

[Sch93] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1), March 1993.

[Sin89] A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989.

[Sin93] A.K. Singh. Program refinement in fair transition systems. *Acta Informatica*, (30):503–535, May 1993.

[SKR91] K. Scneider, T. Kropf, and Kumar R. Integrating a first-order automatic prover in the hol environment. In M Archer, Joyce J.J., Levitt K.N., and Windley P.J., editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications.* IEEE Computer Society Press, 1991.

[Sta93] M.G. Staskauskas. Formal derivation of concurrent programs: An exanple from industry. *IEEE Transaction on Software Engineering*, 19(5):503–528, 1993.

[Taj77] W.D. Tajibnapsis. A correctness proof of a topology information maintenance for a distributed computer network. *Communication of the ACM*, 20(7):477–485, July 1977.

[Tel94] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[UHK94] R. Udink, T. Herman, and J. Kok. Compositional local progress in unity. In *proceeding of IFIP Working Conference on Programming Concepts, Methods and Calculi,*, 1994.

[UK93a] R.T Udink and J.N. Kok. On the relation between unity properties and sequences of states. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Lecture Notes in Computer Science 666, Semantics: Foundations and Applications*, pages 594–608. Springer-Verlag, 1993.

[UK93b] R.T. Udink and J.N. Kok. Two fully abstract models for unity. In E. Best, editor, *Lecture Notes in Computer Science 715, CONCUR'93, Proceedings of the 4th International Conference on Concurrency Theory*, pages 339–352. Springer-Verlag, 1993.

[Zwi88] J. Zwiers. *Compositionality, Concurrency and Partial Correctness.* PhD thesis, Technische Universiteit Eindhoven, 1988.

Index

Samenvatting

HET schrijven van een programma is vaak veel gemakkelijker dan te bewijzen dat een programma doet wat het moet doen. Daarbij wordt onze samenleving steeds afhankelijker van computers; men denke daarbij aan complexe programmatuur die de verkeersleiding van treinen of vliegtuigen regelt, die de administratie van de effectenhandel bijhoudt of die besluit of de stormvloedkering al dan niet neergelaten moet worden.

Om de kwaliteit van programmatuur te kunnen garanderen, wordt ze uitgebreid getest. Dit wordt meestal gedaan met de *probeer-en-faal* methode, waarbij een programma op testinvoer wordt losgelaten en er vervolgens wordt gekeken of het aan de verwachtingen van de programmeur en klant voldoet. Deze manier van testen elimineert helaas niet alle fouten. We horen dan ook vaak mensen over *bugs* in programma's praten. Softwarefabrikanten komen regelmatig met nieuwe versies van hun produkten die de bugs van de vorige versie niet meer bevatten. Dat klopt vaak wel, maar helaas is het eerder regel dan uitzondering dat de nieuwere versie nieuwe bugs bevat. Kortom, bugs lijken als een soort onvermijdbare erfenis in programma's voor te komen. Voor een programma zoals een tekstverwerking kunnen we best wel met een bug of twee leven, maar van een studenten-administratiesyteem verwachten we toch dat het programma niet onbedoeld met de cijfers van de studenten omgaat. We verwachten ook, om een paar andere voorbeelden te noemen, dat onze privé brieven die elektronisch via het Internet worden verzonden niet per ongeluk aan het verkeerde adres worden afgeleverd, of dat een elektronisch besturingsysteem van een vliegtuig niet plotseling weigert. Met de genoemde probeer-en-faal testmethode kunnen wij slechts constateren dat tijdens een test alles goed ging. Helaas is het veelal onmogelijk alle toestanden waarin een computersysteem zich kan bevinden na gaan met een test. Dat zou veel te veel tijd kosten of is soms zelfs theoretisch niet uitvoerbaar, nog daargelaten of het commercieel acceptabel zou zijn.

Een andere methode die tot een betere softwarekwaliteit kan leiden is de volledig formele aanpak; hierbij is een programmeur verplicht om een wiskundig bewijs van de correctheid van zijn programma te leveren. Meestal wordt een programma hierbij ontworpen tegelijkertijd met het construeren van een bewijs dat het programma aan de specificatie voldoet. Er wordt gebruik gemaakt van een speciaal soort wiskunde (vaak *programma-logica* genoemd) voor het redeneren over eigenschappen van een programma. Met deze methode kan men bewijzen dat een programma qua ontwerp foutloos is, zonder dat men het programma zelf hoeft te testen. Dit betekent nog niet dat in de praktijk het geproduceerde programma echt foutloos zal zijn, want het ontwerp is maar een van de vele stadia —alhoewel een van de belangrijkste— in het produceren van software. Wel kunnen wij zeggen dat het resultaat betrouwbaarder zal zijn.

Net zoals dat wij fouten kunnen maken bij het schrijven van een programma, kunnen wij helaas ook fouten maken bij het geven van een bewijs. Een gecompliceerd programma vraagt meestaal ook een gecompliceerd bewijs. De kans om fouten te maken zal toenemen, en het is niet onwaarschijnlijk dat een denkfout die gemaakt wordt bij het ontwerpen van een programma ook gemaakt wordt bij het construeren van een correctheidsbewijs voor dat programma.

Parallel met het ontwikkelen van formele methoden, is ook de technologie om bewijzen te kunnen verifiëren met de computer geëvolueerd. Deze technologie noemen we *mechanisch verificatie* en het computerprogramma die dat doet noemen we (niet echt passend) een *stellingbewijzer.* Een stellingbewijzer wordt gebaseerd op een handvol axiomas en bewijsregels. De consistentie en zinvolheid van deze axiomas en bewijsregels zijn veelal al uitgebreid bestudeerd en er bestaat consensus over hun consistentie. Het bewijs van een nieuwe stelling kan alleen worden geconstrueerd door het herhaaldelijk toepassing van de bewijsregels, uitgaande van de axiomas. De juistheid van deze nieuwe stellingen wordt dus afgedwongen door de manier waarop ze worden gebouwd. In veel stellingbewijzers kan men ook nieuwe bewijsregels definieren in termen van reeds bestaande (primitieve) bewijsregels. Deze nieuwe stellingen en bewijstactieken zijn vaak krachtiger dan de ingebouwde en kunnen dus leiden tot kortere en meer inzichtelijke bewijzen.

In dit proefschrift wordt speciaal aandacht besteed aan zogenaamde gedistribueerde programma's. Een gedistribueerd programma is een programma dat bestaat uit samenwerkende componenten —elke component heeft meestal een eigen processor. Zulke programma's worden heel veel gebruikt, bijvoorbeeld in het Internet waarbij computers uit de hele wereld in een groot elektronisch netwerk worden verbonden. Boodschappen van de ene computer moeten, via tussen-computers, worden verstuurd naar de bestemmingscomputer. Op elk van deze tussen-computers draait een component van een routeringsprogramma. Deze routeringsprogramma's hebben kennis van (een gedeelte) van de structuur van het netwerk. Omdat het netwerk voortdurend van vorm verandert (er kunnen verbindingen bijkomen of wegvallen, en er kunnen tussen-computers aangezet en uitgezet worden) moeten deze computers het netwerk zelf gebruiken om samen uit te vinden hoe de globale structuur is. Zo'n routeringsprogramma is een voorbeeld van een gedistribueerd programma.

Omdat het vaak om veel componenten gaat die over, tussen, met en door elkaar werken, is het redeneren over een gedistribueerd programma moeilijk. In dit proefschrift bestuderen we de programma-logica UNITY die speciaal ontworpen is om te redeneren over eigenschappen van gedistribueerde programma's [CM88]. UNITY is klein en simpel, en daarom aantrekkelijk. Toch is programma's ontwerpen met UNITY, in onze ervaring, vaak erg lastig. Er ontbreekt een systematische ontwerpmethodologie, en sommige ontwerptechnieken bleken niet ondersteund te (kunnen) worden. We hebben dus UNITY uitgebreid, vooral om technieken rond het opsplitsen van programma in parallel werkende componenten beter te ondersteunen. Er worden voorbeelden gegeven om te laten zien hoe we de specificatie van een probleem kunnen vereenvoudigen door een geschikte opsplitsing te kiezen.

We besteden daarbij vooral aandacht aan zogenaamde *zelf-stabiliserende*, gedistribueerde programma's. Een zelf-stabiliserend programma is een programma dat het systeem weer in een gewenste toestand kan brengen als het daar, door een externe verstoring, uit geraakt is. Ook als er tijdens dit herstellen weer nieuwe verstoringen optreden is dat geen probleem voor dergelijke systemen. We hebben UNITY uitgebreid met een reeks van stellingen om over zulke programma's te kunen redeneren. We behandelen een groot voorbeeld, het zogenaamde Eerlijk en Herhaaldelijk Toepassing (EHT) programma. Het EHT programma is een gedistribueerd programma dat een bepaalde klasse van problemen zelf-stabiliserend kan oplossen (uitrekenen).

Dit boek is uniek omdat het niet alleen over formele methoden of over het bewijzen van een of ander moeilijk programma gaat, maar omdat vrijwel alle resultaten mechanisch geverifieerd zijn met een stellingbewijzer! Onze ervaring met mechanisch verificatie wordt ook in dit boek beschreven. Mensen die ook met mechanisch verificatie van gedistribueerde programma's willen beginnen zullen veel onderwerpen in dit boek interessant vinden.

Tot slot willen we benadrukken dat het beoefenen van formele methoden vereist dat programmeurs een goed ontwikkelde wiskundige handvaardigheid hebben. Het effectief opereren met een stellingbewijzer is tot nu toe, helaas, slechts voorbehouden aan een handjevol specialisten. Het is begrijpelijk dat de industrie twijfelt aan de economisch waarde van formele methoden. Toch zullen mensen vroeg of laat, naar mate wij steeds afhankelijker worden van computers, ontdekken hoe kwetsbaar ze zijn als er fouten in computers optreden. In de toekomst zullen mensen dus gedwongen zijn om naar stellingbewijzers te kijken. Investeren in de technologie van mechanisch verificatie en in het opleiden van 'formele' programmeurs, is daarom, naar onze mening, geen weg gegooide moeite.

Curriculum Vitae

Ignatius Sri Wishnu Brata Prasetya

8 december 1966
Geboren te Jakarta, Indonesia.

1982-1985
Sekolah Menengah Atas aan SMAN 68 te Jakarta.

1985-1986
Pre-universiteit opleiding aan IIVO te Utrecht.

1986-1988
Propaedeuse Informatica aan Technische Universiteit Delft.

1987-1991
Doctoraal Informatica aan Technische Universiteit Eindhoven.

1991-1995
Assistent in Opleiding aan de Vakgroep Informatica van de Universiteit Utrecht.