Formalizing UNITY with HOL

I.S.W.B. Prasetya [1]
Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB UTRECHT, the Netherlands

[1]current address: Dr Ir I. Wishnu S.B. Prasetya, Faculty of Computer Science, University of Indonesia, Kampus UI Depok, Depok 16424, Indonesia email: `wishnup@caplin.cs.ui.ac.id`

Abstract

THIS paper has been written for the IPA workshop in Egmond aan Zee, 16-20 october 1995. Despite its size, it is intended as an introduction —a quick tour— to the technology of mechanical verification and the formal design of distributed algorithms, and is not intended to be complete. Nevertheless it will provide the necessary information for the reader to understand the topics. For further reading on the topics, the reader can try the introduction book to HOL [GM93] and the book of UNITY [CM88]. Most of this paper is taken from my Ph.D. thesis. If the reader is interested in further technical details, my thesis is available through ftp at: `ftp.cs.ruu.nl` in directory `pub/RUU/CS/phdtheses/Prasetya`

Chapter 1

Introduction

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 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[1] (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,

[1] 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.

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.

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. In this paper we will take a look at HOL, a system developed by M. Gordon [GM93]. HOL is based on Gordon's higher order logic[2]. There are other theorem provers with a more powerful logic (we present HOL in this paper because it is with which we have the most experience). 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.

During our research we use HOL to mechanically verify distributed algorithms. Distributed algorithms are inherently difficult to deal with. Such an algorithm consists of a number of processes that interacts with each other —often in a very subtle way. If we try to reason about it informally, our reasoning will be prone to error. How many times

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

we have heard of this or that distributed algorithms being discovered to be flawed? Using formal approaches not only 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].*

To be able to verify an algorithm —a program— with HOL we need to extend it first with a suitable programming logic. HOL needs to know all the definitions and axioms supported by the logic. The choice of the programming logic will depend on our purpose —on the kind of program properties we want to prove. We have chosen for UNITY. Its simplicity is our main reason for the choice, but in any case, it is our experience with HOL and UNITY that we wish to share with the participants of this workshop. During the workshop there will be other talks in which the participants will be shown how to verify programs with other theorem provers and other programming logics.

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.

The rest of this paper will be organized as follows. In Chapter 2 we will give a brief introduction to the theorem prover HOL. We will explain how it works, how to write formulas in HOL, and how to write a proof in HOL. Chapter 3 will explain the programming logic UNITY. It will be explained what a UNITY program is, and how to reason about its properties. Finally, Chapter 4 will show how to extend HOL with UNITY. It will be sketched how the correctness of a UNITY program can be verified using HOL[3].

[3]The reader should bear in mind that in general it is hard to automatically prove the correctness of a distributed algorithm. But even in the absence of an automatically generated proof, we can use a theorem prover to verify a hand written proof.

Chapter 2

The Theorem Prover HOL

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 proven 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. So, we have actually two levels here: the actual HOL level (or the formula level) and the ML level (or the tools level). The illustration in Figure 2.1 will help to remind the reader.

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.

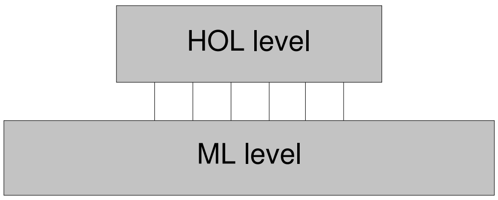

Figure 2.1: Two levels in working with HOL.

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 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.

Having said all these, let us now take a closer at the HOL system.

2.1 Formulas in HOL

Figure 2.2 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.

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.

	standard notation	**HOL notation**
Denoting types	$x \in A$ or $x : A$	`"x:A"`

Proposition logic	$\neg p$, true, false
	$p \wedge q$, $p \vee q$
	$p \Rightarrow q$
		`"~p"`, `"T"`, `"F"`
		`"p /\ q"`, `"p \/ q"`
		`"p ==> q"`

Universal quantification
 	
Existential quantification
	$(\forall x, y :: P)$
	$(\forall x : P : Q)$
	$(\exists x, y :: P)$
	$(\exists x : P : Q)$
		`"(!x y. P)"`
		`"(!y::P. Q)"`
		`"(?x y. P)"`
		`"(?x::P. Q)"`

Function application λ abstraction	$f.x$ $(\lambda x.\ E)$	`"f x"` `"(\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	$x \in V$, $U \subseteq V$
	$U \cup V$, $U \cap V$
	$U \backslash V$
		`"x IN V"`, `"U SUBSET V"`
		`"U UNION V"`, `"U INTER V"`
		`"U DIFF V"`

Lists	$a;s$, $s;a$ $[a; b; c]$, st	`"CONS a s"`, `"SNOC a s"` `"[a;b;c]"`, `"APPEND s t"`

Figure 2.2: The HOL Notation.

As an example:

```
"{ (\x:*A. (x=X) => 0 | (f x)) x | ((g:*A->*B) x) IN (A INTER B)}"
```

is a HOL-term representing the set $\{f'.x \mid g.x \in A \cap B\}$ where f' is a function such that $f'.x = 0$ if $x = X$ and else $f'.x = f.x$. Note that the bound variable `x` ranges over the polymorphic type `*A`.

2.1.1 Types

More precisely, there are four kinds of type in HOL. First, there are *type variables* such as `*A` above to represent polymorphic types. Then we have *atomic types*, for example `bool` and `ind` (representing the Boolean set of true and false, and the infinite set). Third, we have compound types which have the form of $(\sigma_1, \sigma_2, \ldots)op$ where σ_i's are types. op is called a *type operator*. For example, `*A#*B` is the product type of the type `*A` and type `*B`. `(num)list` is the type of lists over `num`. And finally we have *function types*. It is denoted like, for example, `*A->*B` which represent the set of all *total* functions[1] from `*A` to `*B`.

We can define new types operators. For example we can define the type operator `triple` as follows:

```
define_type `triple_DEF` `triple = TRIPLE *A *B *C`
```

[1] All functions in HOL are required to be total.

This defines a new type operator `triple` which have three arguments. For example, `"x:(num,bool,bool)triple"` is now a valid HOL-term. The type `(num,bool,bool)` consists of all elements of the form `TRIPLE x y z` where `x` is of type `num`, and `x` and `y` are of type `bool`.

We can also define recursive types. For example, here is how we can define type operator `list` (which represent the set of lists):

```
define_type `list_DEF` `list = NIL | CONS * list`
```

Now `CONS 1 (CONS 2 (CONS 2 NIL))` is a valid HOL-term (in HOL, `NIL` is also written `[]` and `CONS x (CONS y (CONS z NIL))` is also written `[1;2;3]`). In addition, `define_type` also generates a definitional theorem stating the isomorphism between the newly defined type with some subset of existing types. For example the above type definition will generate the following theorem:

```
|- !e f. ?! fn. (fn NIL = e) /\ (!x l. fn(CONS x l) = f(fn l)x l)
```

and the theorem will be called `list_DEF`.

2.2 Theorems in HOL

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`. So far, we have seen an example of a theorem. Here is another one:

```
"P 0 /\ (!n. P n ==> P (SUC n)) |- (!n. P n)"
```

which is the induction theorem on natural numbers[2].

The core of HOL system consists of five axioms. Theorems can be derived from these axioms using HOL (primitive) rules. The first four axioms are:

```
BOOL_CASES_AX   :  |- !t. (t=T) \/ (t=F)
IMP_ANTISYM_AX :  |- !t1 t2. (t1 ==> t2) ==> (t2 ==> t1) ==> (t1=t2)
ETA_AX          :  |- !t. (\x. t x) = t
SELECT_AX       :  |- !(P:*->bool) x. P x ==> P ($@ P)
```

[2] All variables which occur free are assumed to be either constants or universally quantified.

where `@` denotes the choice operator. `$@ P` picks an element x such that it satisfies `P` —if such an element exists (which is exactly what the forth axiom above says). As for the other axioms: the first states that a boolean term is either true or false[3]; the second axiom states that $\Rightarrow$ is anti-symmetric; and the third states that $(\lambda x.f.x) = f$.

The fifth axiom states that there exists a bijection between the type `ind` and itself, but the bijection is not surjective. In otherwords, the type `ind` is thus infinite. The axiom is given below:

```
INFINITY_AX    :  |- ?f:ind->ind. ONE_ONE f /\ ~(ONTO f)
```

HOL has 8 primitives rules:

i. `ASSUME`. It introduces an assumption: `ASSUME "t"` generates the theorem `t |- t`.

ii. `REFL`. It yields a theorem stating that any HOL-term `t` is equal to itself: `REFL "t"` generates `|- t=t`.

iii. `BETA_CONV`. It does Beta reduction. For example, `BETA_CONV "(\x. (x + 1)) X"` generates `|- (\x. (x+1)) X = (X+1)`.

iv. `SUBST`. It is used to perform a substitution within a theorem.

v. `ABS`. For example, `ABS "t1 = t2"` generates a theorem `|- (\x. t1) = (\x. t2)`.

vi. `INST_TYPE`. It is used to instantiate type variables in a theorem.

vii. `DISCH`. It is used to discharge an assumption. For example, `DISCH (A1; A2 |- t)` yields `A1 |- A2 ==> t`.

viii. `MP`. It applies the Modus Ponens principle. For example, `MP (|- t1 ==> t2) (|- t1)` generates `|- t2`.

More sophisticated rules can be constructed by combining the above primitive rules. Some examples of derived rules which are frequently used 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` is a smarter version of `MP` (both apply the modus ponens principle). Below are some examples of HOL sessions.

```
1 #DE_MORGAN_THM ;;
2 |- !t1 t2. (~(t1 /\ t2) = ~t1 \/ ~t2) /\ (~(t1 \/ t2) = ~t1 /\ ~t2)
3
4 #th1 ;;
5 |- ~(p /\ q) \/ q
6
7 #REWRITE_RULE [DE_MORGAN_THM] th1 ;;
8 |- (~p \/ ~q) \/ q
```

◀

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

[3] Hence the HOL logic is conservative.

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.

2.3 Extending HOL

The core of HOL provides predicate calculus. To use it for a particular purpose, we still need to extend it. For example, if we want to use it to verify programs, we need first to define what programs and specifications are. There two ways to extend HOL: by adding axioms or by adding definitions. Adding axioms is considered dangerous because we can introduce inconsistency. While it is still possible to introduce absurd definitions, they are nothing more than abbreviations, and hence cannot introduce inconsistency. Definitional extension is therefore a much preferred practice.

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. 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).

As a side note, 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.*

2.4 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 2.3. 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.

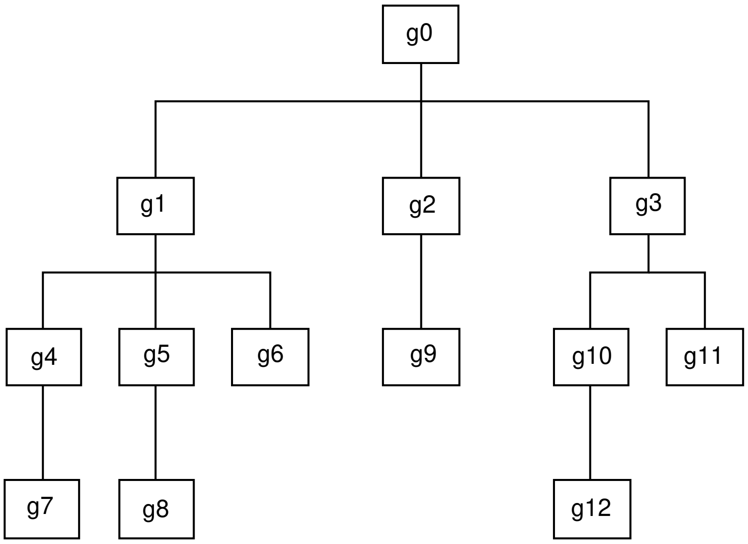

Figure 2.3: A proof tree.

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 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

```
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 2.4: An example of an interactive backward proof in HOL.

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. 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 2.4.

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`[4] 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 ])
```

◀

2.5 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

[4] The name of the theorem defining the constant `MAP` happens to have the same name. These two `MAP`s really refer to different things.

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[5] 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[6]. 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].

[5]In general, natural number arithmetic is not decidable if multiplication is included. So the best we can achieve is a partial decision procedure.

[6]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 3

The Programming Logic UNITY

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]. In this chapter we will take a closer to UNITY. There are of course pros and contras for UNITY, but let us not discuss them here. We will show here how things can be done with UNITY, and leave it to the reader to judge what kinds of applications he wants to use the logic for.

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. 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.

3.1 State, Predicates, and Actions

First, let us give a brief review on some basic notions in programming. A program has a set of variables. The values of these variables at a given moment is called the *state* of that program at that moment. Let us assume the universe Var of *all* program variables, and Val of all values the variables may take. A program *state* can be represented by a function $s \in \mathsf{Var}{\rightarrow}\mathsf{Val}$. The set of all (program) states is denoted by State.

A predicate over A is a function from A to $\mathbb{B}$. It characterizes a subset of A. A *state-predicate* is a predicate over State. For example, state-predicates are used to specify pre and post conditions of a program.

The set of all state-predicates is denoted with Pred. Standard Boolean operators ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\exists$, $\forall$) can be lifted to the predicate levels. For example, for all predicates p and q, we can define $\neg p = (\lambda s.\neg p.s)$ and $p \wedge q = (\lambda s.p.s \wedge q.s)$. The lifting preserves all algebraic properties of these operators.

A predicate p over A is said to hold *everywhere* —denoted usually by $[p]$— if $p.s$ holds for all $s \in A$.

An action (statement) of a program can change the state of a program. An obvious way to represent an action is by a function a where the state resulting from the execution of a on a state s is given by $a.s$. Such an action is however always deterministic. To allow non-determinism, we will represent an action as a relation on State. That is, an action a has the type $\mathsf{State}{\rightarrow}\mathsf{State}{\rightarrow}\mathbb{B}$. The interpretation of $a.s.t$ is that t is a possible state resulting from executing a at state s. However, if t is the only posible final state, then a will end up with t. We can define Hoare triple as follows:

$$\{p\}\ a\ \{q\} \;=\; (\forall s,t :: p.s \wedge a.s.t \Rightarrow q.t) \tag{3.1.1}$$

All kind of basic laws for Hoare triples are derivable from this definition. For example we have:

$$\frac{(\{p\}\ a\ \{q\}) \wedge (\{r\}\ a\ \{s\})}{\{p{\wedge}r\}\ a\ \{q{\wedge}s\}} \quad \text{and} \quad \frac{[p{\Rightarrow}q] \wedge (\{q\}\ a\ \{r\}) \wedge [r{\Rightarrow}s]}{\{p\}\ a\ \{s\}}$$

The set of all action will be called Action.

In practice, because the variables of a program is (much) less that what are available in Var, we only have to consider a more restricted state space. Given a set of variables V, a state-predicate p is said to be *confined* by V, denoted $p \in \mathsf{Pred}.V$, if p is a predicate over $V{\rightarrow}\mathsf{Val}$. Such a predicate only contains information on the values of variables in V.

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

Figure 3.1: The program Fizban

3.2 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 3.1 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. The init section describes the assumed initial states of the program. In the program Fizban in Figure 3.1, the initial condition is true, which means that the program may start in any state. The assign section lists the actions (statements) 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: *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.

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 $A \subseteq \mathsf{Action}$ is a set consisting of the actions of P, $J \in \mathsf{Pred}$ is a predicate describing the possible initial states of P, and $V_r, V_w \in \mathsf{Var}$ 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 3.2.1 Uprog DESTRUCTORS

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

◀

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

$$\mathbf{i}P = \mathbf{r}P \backslash \mathbf{w}P \tag{3.2.1}$$

3.2.1 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][1].

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

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

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

[1]We omit the always section and split the declare section into read and write parts

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. 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.

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 seem to be reasonable requirements. 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 3.2.2 Unity

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

◀

The specific definition of AlwaysEn, $\not\leftarrow$, and $\not\rightarrow$ is unimportant now. Suffices here to say that $(\forall a : a \in \mathbf{a}P : \Box_{\mathsf{En}} a)$ means that for any action in P, if it is guarded, it will behave as skip if the guard fails. The condition $(\mathbf{w}P)^{\mathsf{c}} \not\leftarrow a$ means that all variables not declared as write variables of P (hence outside $\mathbf{w}P$) are ignored by a (hence a only writes to the declared write variables). The condition $(\mathbf{r}P)^{\mathsf{c}} \not\rightarrow a$ means that all variables not declared as read variables cannot influence the effect of a (hence a only depends on the declared read variables).

3.2.2 Parallel Composition

There are several ways to compose or to transform programs in UNITY. For example, we can add assignments on fresh variables, strengthen guards, or combine two programs in parallel. In this paper, we will only touch the topic of parallel composition, because this is the most interesting composition in distributed programming. If the reader is interested, see for example [CM88, R.95, Pra95].

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 3.2.3 PARALLEL COMPOSITION

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

◄

As an example, we can compose the program Fizban in Figure 3.1 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 3.2.1, 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.

3.3 Programs' Behavior

To facilitate reasoning about program behavior UNITY provides several primitive operators. The discussion in Section 3.2 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 induce some progress. For example, the execution of the action a_4 in Fizban $\|$ TikTak will establish $x = 0$ regardless when it is executed. 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.

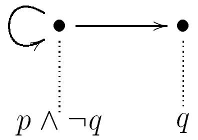

Figure 3.2: p unless q: the predicates $p \wedge \neg q$ and q define sets of states. The arrows depict possible transitions between the two sets of states.

Definition 3.3.1 UNLESS

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

Definition 3.3.2 ENSURES

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

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 3.2 may be helpful.

If there exists an action a that can do the horizontal transition in Figure 3.2, that is a transition from $p \wedge \neg q$ to q. Notice that the diagram tells us that as long as q is not established, we will remain in $p \wedge \neg q$. However now, by the fairness assumption of UNITY, a will eventually be executed and hence q will be established. Hence, we have progress from p to q and this is what p ensures q means.

As an example, the program Fizban in Figure 3.1, which has the following assign section:

$$\begin{array}{ll} & \mathsf{if}\ a = 0\ \mathsf{then}\ x := 1 \\ [\!] & \mathsf{if}\ a \neq 0\ \mathsf{then}\ x := 1 \\ [\!] & \mathsf{if}\ x \neq 0\ \mathsf{then}\ y, x := y + 1, 0 \end{array}$$

satisfies the following properties:

$$_{\mathsf{Fizban}}\vdash \quad (a = X)\ \mathsf{unless\ false} \tag{3.3.1}$$

$$_{\mathsf{Fizban}}\vdash \quad \mathsf{true\ unless}\ (x = 1) \tag{3.3.2}$$

$$_{\mathsf{Fizban}}\vdash \quad (a = 0)\ \mathsf{ensures}\ (x = 1) \tag{3.3.3}$$

$$_{\mathsf{Fizban}}\vdash \quad (a \neq 0)\ \mathsf{ensures}\ (x = 1) \tag{3.3.4}$$

If (3.3.1) holds for any X then it states that Fizban cannot change the value of a. (3.3.2) is an example of a property that trivially holds in any program (the reader can check it by unfolding the definition of unless). (3.3.3) and (3.3.4) describe single-action progress from, respectively $a = 0$ and $a \neq 0$ to $x = 1$.

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 3.3.3 STABLE PREDICATE

$$_P\vdash \circlearrowleft p \;=\; _P\vdash p \;\textsf{unless false}$$

◀

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

$$_P\vdash \circlearrowleft p \;=\; (\forall a : a \in \mathbf{a}P : \{p\}\ a\ \{p\}) \tag{3.3.5}$$

Consequently, if p holds initially and is stable in P, it will hold throughout any execution of P, and hence it is an *invariant*.

In Figure 3.3 is a list of some interesting properties of unless and $\circlearrowleft$—there are more though, see for example [CM88]. The properties can be derived from the definition. Some of the properties (Theorems 3.3.5 and 3.3.7) look similar to some well known rules for Hoare triples[2].

3.3.1 A More General Progress Operator

The ensures operator is still too restricted to describe progress (it only describes single action progress). Intuitively, progress seems to have transitivity and disjunctivity properties. For example, if a system can progress from p to q and from q to r, then it can progress from p to r (transitivity). If it can progress from p_1 to q and p_2 to q, then from either $p_1 q$ or p_2 it can progress to q (disjunctivity). As a more general progress operator we can therefore take the smallest closure of ensures which is transitive and disjunvtive. The resulting operator is the *leads-to* operator, denoted by $\mapsto$:

Definition 3.3.10 LEADS-TO $(\lambda p, q.\ _P\vdash p \mapsto q)$ is defined as the smallest relation R satisfying:

i. $$\frac{_P\vdash p \;\textsf{ensures}\; q}{R.p.q}$$

ii. $$\frac{R.p.q \wedge R.q.r}{R.p.r}$$

iii. $$\frac{(\forall i : i \in W : R.(p_i).q)}{R.(\exists i : i \in W : p_i).q}$$

◀

Obvious properties of $\mapsto$ is that it satisfies ***i***, ***ii***, and ***iii*** above. $_P\vdash p \mapsto q$ implies that that if p holds during an execution of P, then eventually q will hold, so it corresponds with our intuitive notion of progress. The problem with this operator is that it is not very compositional. In designing a program, we often split our program into smaller

[2] Note though that the pre-condition strengthening principle of Hoare triples does not apply to unless

Theorem 3.3.4 unless INTRODUCTION

$$\frac{[p \Rightarrow q] \;\vee\; [\neg p \Rightarrow q]}{{}_{P}\vdash p \;\mathsf{unless}\; q}$$

Theorem 3.3.5 unless POST-WEAKENING

$$\frac{({}_{P}\vdash p \;\mathsf{unless}\; q) \;\wedge\; [q \Rightarrow r]}{{}_{P}\vdash p \;\mathsf{unless}\; r}$$

Theorem 3.3.6 unless SIMPLE CONJUNCTION

$$\frac{({}_{P}\vdash p \;\mathsf{unless}\; q) \;\wedge\; ({}_{P}\vdash r \;\mathsf{unless}\; s)}{{}_{P}\vdash p \wedge r \;\mathsf{unless}\; q \vee s}$$

Theorem 3.3.7 unless SIMPLE DISJUNCTION

$$\frac{({}_{P}\vdash p \;\mathsf{unless}\; q) \;\wedge\; ({}_{P}\vdash r \;\mathsf{unless}\; s)}{{}_{P}\vdash p \vee r \;\mathsf{unless}\; q \vee s}$$

Theorem 3.3.8 ⟳ CONJUNCTION

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

Theorem 3.3.9 ⟳ DISJUNCTION

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

Figure 3.3: Some properties of unless and ⟳.

Theorem 3.3.11 unless COMPOSITIONALITY

$$({}_{P}\vdash p \text{ unless } q) \ \wedge \ ({}_{Q}\vdash p \text{ unless } q) \quad = \quad ({}_{P\|Q}\vdash p \text{ unless } q)$$

Theorem 3.3.12 $\circlearrowright$ COMPOSITIONALITY

$$({}_{P}\vdash \circlearrowright J) \ \wedge \ ({}_{Q}\vdash \circlearrowright J) \quad = \quad ({}_{P\|Q}\vdash \circlearrowright J)$$

Theorem 3.3.13 ensures COMPOSITIONALITY

$$\frac{({}_{P}\vdash p \text{ ensures } q) \ \wedge \ ({}_{Q}\vdash p \text{ unless } q)}{{}_{P\|Q}\vdash p \text{ ensures } q}$$

Figure 3.4: Some theorems to combine the properties of parallel components.

components (*modularity* principle). In doing so, we must be able to split the specification of a program into the specifications of the components. If a property a program is not (very) compositional, it will be hard to split it into properties of component programs and hence we will not be able to do component-decomposition during our design. In Figure 3.4 are several theorems which will enable us to combine the properties of component programs.

In particular, notice how Theorem 3.3.13 describes the condition in which progress by ensures can be preserved by parallel composition. The theorem does not apply to progress by $\mapsto$ though. In our research, we use a variant of $\mapsto$ called *reach* operator, denoted by $\rightarrowtail$. This operator is more compositional that $\mapsto$. For example, progress by $\rightarrowtail$ cannot be destroyed by parallel composition of programs that do not share write variables. We will return to this topic later. In the sequel, we will simply abandon the $\mapsto$ operator (which is the 'standard' progress operator in UNITY) and use the $\rightarrowtail$ operator.

Definition 3.3.14 REACH OPERATOR $(\lambda p, q. J \;{}_P\!\vdash p \rightarrowtail q)$ is defined as the smallest relation R satisfying:

i. $$\frac{p, q \in \mathsf{Pred}.(\mathbf{w}P) \;\wedge\; ({}_P\!\vdash \circlearrowleft J) \;\wedge\; ({}_P\!\vdash J \wedge p \;\mathsf{ensures}\; q)}{R.p.q}$$

ii. $$\frac{R.p.q \wedge R.q.r}{R.p.r}$$

iii. $$\frac{(\forall i : i \in W : R.(p_i).q)}{R.(\exists i : i \in W : p_i).q}$$

where W is assumed to be non-empty.

◀

Intuitively, $J \;{}_P\!\vdash p \rightarrowtail q$ implies that J is stable in P and that P can progress from $J \wedge p$ to q. In addition, the type of p and q is restricted: they are predicates over $\mathbf{w}P \rightarrow \mathsf{Val}$ (the part of state space restricted to the write variables of P). If we think it over, since P can only write to its write variables, whatever progress it may make, it will make it on these variables. This is why we think it is reasonable to restrict the type of p and q as above. Whatever values variables outside $\mathbf{w}P$ may have will remain stable then, and can therefore be specified in J. This division turns out to yield a compositional progress operator. The operator $\rightarrowtail$ satisfies ***i***, ***ii***, and ***iii*** in Definition 3.3.14. Figure 3.5 list some other interesting properties of $\rightarrowtail$. $\rightarrowtail$ INTRODUCTION states that $p \Rightarrow q$ then it is trivial that any program P can progress from p to q. $\rightarrowtail$ DISJUNCTION states that $\rightarrowtail$ is disjunctive at its left and right operands. SUBSTITUTION states that, like Hoare triples, we can strengthen pre-conditions and weaken post-conditions. The PSP law states how we a safety property (unless) of a program can influence its progress. STABLE STRENGTHENING states that we can also strengthen the J-part of a $\rightarrowtail$ specification with another stable predicate[3].

3.4 An Example: Leader Election

As an example, let us consider a derivation of a 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. That is, if the program is executed several times with the same initial states, it should not be the case that it keeps selecting the same leader.

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:

$$i^+ \;=\; (i+1) \;\mathsf{mod}\; N$$

[3] We are not allowed to weaken the J-part though.

Theorem 3.3.15 $\rightarrowtail$ Introduction

$$\frac{p, q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\ _P\!\vdash \circlearrowright J) \ \wedge \ [J \wedge p \Rightarrow q]}{J \ _P\!\vdash p \rightarrowtail q}$$

Theorem 3.3.16 $\rightarrowtail$ Disjunction

$$\frac{(J \ _P\!\vdash p \rightarrowtail q) \ \wedge \ (J \ _P\!\vdash r \rightarrowtail s)}{J \ _P\!\vdash p \vee r \rightarrowtail q \vee s}$$

Theorem 3.3.17 $\rightarrowtail$ Substitution

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

Theorem 3.3.18 $\rightarrowtail$ Progress Safety Progress (PSP)

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

Theorem 3.3.19 $\rightarrowtail$ Stable Strengthening

$$\frac{(\ _P\!\vdash \circlearrowright J_2) \ \wedge \ (J_1 \ _P\!\vdash p \rightarrowtail q)}{J_1 \wedge J_2 \ _P\!\vdash p \rightarrowtail q}$$

Figure 3.5: Some properties of $\rightarrowtail$.

◀

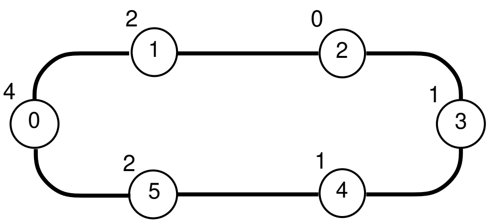

Figure 3.6: A ring network.

Figure 3.6 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 3.6 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'.

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:

$$\mathsf{LS0}: \quad \mathsf{true} \;{}_{\mathsf{ring}}\!\vdash \mathsf{true} \rightarrowtail \mathsf{Ok}$$

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 that by its definition, the $\rightarrowtail$ relation is transitive and disjunctive. There is a law called Bounded Progress law that states that any transitive and left-disjunctive relation $\rightarrow$ satisfies:

$$\textsc{Bounded Progress}: \quad \frac{q \rightarrow q \;\wedge\; (\forall M :: p \wedge (m = M) \rightarrow (p \wedge (m \prec M)) \vee q)}{p \rightarrow q}$$

if $\prec$ is well-founded. Suppose that from p we can either preserve p while decreasing m, or we can go to q. The well-foundedness of $\prec$ will prevent us from keep decreasing m, and hence, when m cannot be decreased anymore, we will have to go to q. This is what the law above states.

The previously described strategy (either Ok is established or $x.0$ decreases) is an instance of Bounded Progress. Let us now apply the principle to $\mathsf{LS0}$:

$$
\begin{array}{ll}
 & \mathsf{true} \vdash \mathsf{true} \rightarrowtail \mathsf{Ok} \\
\Leftarrow & \{ \text{ the Bounded Progress principle } \} \\
 & (\forall M :: \mathsf{true} \vdash (x.0 = M) \rightarrowtail (x.0 < M) \vee \mathsf{Ok}) \;\wedge\; (\mathsf{true} \vdash \mathsf{Ok} \rightarrowtail \mathsf{Ok}) \\
\Leftarrow & \{ \text{ Definition of } \rightarrowtail \} \\
 & (\forall M :: \mathsf{true} \vdash (x.0 = M) \rightarrowtail (x.0 < M) \vee \mathsf{Ok}) \;\wedge\; \mathsf{Ok} \in \mathsf{Pred}.(\mathbf{w}(\mathsf{ring})) \;\wedge \\
 & (\;\vdash \mathsf{Ok}\ \mathsf{ensures}\ \mathsf{Ok}) \\
= & \{ \; p\ \mathsf{ensures}\ p \text{ always hold } \} \\
 & (\forall M :: \mathsf{true} \vdash (x.0 = M) \rightarrowtail (x.0 < M) \vee \mathsf{Ok}) \;\wedge\; \mathsf{Ok} \in \mathsf{Pred}.(\mathbf{w}(\mathsf{ring}))
\end{array}
$$

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:

$$\mathsf{ts} \;=\; N \;-\; \mathsf{max}\{n \mid (n \leq N) \wedge (\forall i : i < n : x.i = x.0)\} \tag{3.4.1}$$

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):

$$
\begin{array}{ll}
 & \mathsf{true} \vdash (x.0 = M) \rightarrowtail (x.0 < M) \vee \mathsf{Ok} \\
\Leftarrow & \{ \text{ the Bounded Progress principle } \} \\
 & (\forall K : K < N : \mathsf{true} \vdash \;\; (x.0 = M) \wedge (\mathsf{ts} = K) \rightarrowtail \\
 & \qquad\qquad ((x.0 = M) \wedge (\mathsf{ts} < K)) \vee \mathsf{Ok} \vee (x.0 < M)) \\
\Leftarrow & \{ \text{ definition of } \rightarrowtail \} \\
 & (\forall K : K < N : \;\vdash\;\; (x.0 = M) \wedge (\mathsf{ts} = K)\ \mathsf{ensures} \\
 & \qquad\qquad ((x.0 = M) \wedge (\mathsf{ts} < K)) \vee \mathsf{Ok} \vee (x.0 < M))
\end{array}
$$

The resulting specification above 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.

So, to summarize, we come to the following refinement of LS0:

```
prog    ring
read    {x.i | i < N}
write   {x.i | 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 3.7: Leader election in a ring.

◀

For all $M \in \mathbb{N}$ and $K < N$:

LS1.a: $\{x.i \mid i < N\} \subseteq \mathbf{w}(\mathsf{ring})$

LS1.b: $(x.0 = M) \wedge (\mathsf{ts} = K)$
$\quad$ ensures $((x.0 = M) \wedge (\mathsf{ts} < K)) \vee \mathsf{Ok} \vee (x.0 < M)$

◀

Without further proof, a program that satisfies the above specifications is presented in Figure 3.7[4]. Notice how the non-determinism in the identity of the selected leader relies on the non-determinism in the ordering in which the actions of the program in Figure 3.7 are executed during a parallel execution.

3.5 Progress under Parallel Composition

We said earlier that the reason that we use the reach operator $\rightarrowtail$ instead of the standard leads-to operator is that because the leads-to operator is not compositional. For the sake of completeness we will show now some most important compositionality laws on $\rightarrowtail$.

Let us first define a special case of unless as follows:

Definition 3.5.1 unless_V
Let V be a set of variables:

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

Note that s and X range over states.

◀

In particular, if $V = \mathbf{r}P \cap \mathbf{w}Q$ (hence V are the 'border' variables from Q to P), $_Q\vdash p\ \mathsf{unless}_V\ q$ means that under condition p, Q *cannot* influence P without establishing q.

[4]In the read and write sections of the program in Figure 3.7 "$\{x.i \mid i < N\}$" denotes a set of variables. Another notation which the reader is perhaps more familiar with is: x: array $[0 \ldots N)$ of Val

So, for example, ${}_Q\vdash p$ unless_V false means that Q cannot influence P as long as p holds; ${}_Q\vdash$ true unless_V q means that Q always marks its interference to P by establishing q.

Let $V = \mathbf{r}P \cap \mathbf{w}Q$. Suppose under condition Q cannot influence P without establishing s (hence ${}_Q\vdash$ true unless_V q). Q cannot then destroy any progress in P without establishing q. This principle —or actually, a more general version thereof— is formulated by the law below. It is called *Singh* law.

Theorem 3.5.2 SINGH LAW

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

where $V = \mathbf{r}P \cap \mathbf{w}Q$.

◀

A corollary of Singh Law is given below. Compare it with the compositionality law of $\mathsf{ensures}$ (Theorem 3.3.13).

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

$\rightarrowtail$ also satisfies a very nice principle called the *Transparency* principle:

Theorem 3.5.3 TRANSPARENCY LAW

$$\frac{(\mathbf{w}P \cap \mathbf{w}Q = \emptyset) \;\wedge\; ({}_Q\vdash \circlearrowright J) \;\wedge\; (J \;{}_P\vdash p \rightarrowtail q)}{J \;{}_{P\|Q}\vdash p \rightarrowtail q}$$

◀

So, in a network of components with disjoint write variables, any progress $J \;{}_P\vdash p \rightarrowtail q$ in a component P will be preserved if all components respect the stability of J. A network of programs with disjoint write variables occur quite often in practice. For example a network of programs that communicate using channels can be modelled a network with write-disjoint components.

Chapter 4

Embedding UNITY in HOL

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. 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:

```
cond (\s. s x < s y) (assign x (\s. f (s x))) (assign x (\s. 0))
```

In Chapter 2 we have briefly introduced the reader to the theorem prover HOL. In Chapter 3 we have discussed the programming logic UNITY which is a tool to formally reason about distributed programs. But even with formal method complicated programs are still complicated to prove, and we will feel more comfortable if their proofs can also

[1]However, adding axioms, as remarked before, is not a recommended practice.

be mechanically checked by a theorem prover such as HOL. So the question now 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]. Some laws were shown in the previous chapter (more can be found in [CM88, Pra95]). 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. 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.

4.1 Representing a Program in HOL

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`.

[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]

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[3].

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 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" ;;
```

4.1.1 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:

[3]This means however 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.

HOL-definition 4.1.1

```
|- 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 laws given in Chapter 3 is $p \in \mathsf{Pred}.V$, which means that p is a predicate over $V \rightarrow \mathsf{Val}$. The type $V \rightarrow \mathsf{Val}$ is a sub-type of $\mathsf{Var} \rightarrow \mathsf{Val}$. In HOL it is unfortunately not so easy to define a sub-type. So instead, we will represent an object s of type $V \rightarrow \mathsf{Val}$ by an object s' of type $\mathsf{Var} \rightarrow \mathsf{Val}$. The value of $s'.v$ for $v \notin V$ can be considered irrelevant. So this is how we are going to define $p \in \mathsf{Pred}.V$ (we call this V *confines* p):

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

where the function projection $\upharpoonright$ is defined as:

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

Predicate confinement is defined as follows in HOL:

HOL-definition 4.1.2

```
|- !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))
```

◀

4.1.2 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, an assignment $v := E$ can be defined as follows in HOL:

```
let Assign_DEF = new_definition
  (`Assign_DEF`,
  "(Assign v E):^Action = (\s t. (!x. (x=v) => (t v = E s) | (t v = s v)))") ;;
```

[4]Some people prefer to use functions instead of relations. If functions are used, then actions are deterministic.

So, an assignment $x := x + 1$ can be represented by `Assign x (\s. (s x) + 1)`.

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:

HOL-definition 4.1.3

```
|- !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)
```

◀

As an example of the embedding/representation of a UNITY program, consider the distributed program below. The program computes the simple minimal distance between any pair of nodes in a network. The network consists of a set V of nodes. Each node a has a set of of neighbors given by $N.a$.

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 4.1 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.

[5] A better set library is under development.

```
let MinDist = new_definition
  (`MinDist`,
   "MinDist d (V:*node, N) =
     ( (??a::(\a. a IN V). (\act. act = Assign (d a a) (\s. 0))) OR
       (??a::(\a. a IN V).
          (??b::(\b. b IN V /\ ~(a=b)).
                (\act. act = Assign (d a b) (\s:*var->num. MIN {(s (d a b')) + 1 | b' IN (N b)})))),
        TT,
       (??a::(\a. a IN V). (??b::(\b. b IN V). (\v. v = d a b))),
       (??a::(\a. a IN V). (??b::(\b. b IN V). (\v. v = d a b))) )") ;;
```

Figure 4.1: The HOL definition of the program MinDist.

In Chapter 3 we have defined a predicate Unity to characterizes all well-formed UNITY programs. The definition is re-displayed below:

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

The first and the second condition are obvious. The third is intended to mean that all actions should be always-enabled (that is, a transition is always possible from any state). The fourth should mean that variables not declared as write variable of P are ignored by P, hence they cannot be written. The last conjunct should mean 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.

We have not yet given the definition of $\square_{\mathsf{En}}$, $\not\leftarrow$, and $\not\twoheadrightarrow$. A precise definition of these notions is required. Otherwise we may not be able to derive various compositionality results such as listed in Section 3.5. Below are the exact definition that we use for them:

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

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

iii. $V \not\twoheadrightarrow a = (\forall s,t,s',t' :: \left(\begin{array}{c}(s \restriction V^{\mathsf{c}} = s' \restriction V^{\mathsf{c}}) \wedge (t \restriction V^{\mathsf{c}} = t' \restriction V^{\mathsf{c}}) \\ \wedge (s' \restriction V = t' \restriction V) \wedge a.s.t\end{array}\right) \ \Rightarrow \ a.s'.t')$

and the corresponding HOL definition:

HOL-definition 4.1.4

```
|- !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 4.1.5

```
|- !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 4.1 can be shown to satisfy the predicate `UNITY` above.

4.2 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 $\rightarrowtail$, which is some sort of transitive and left-disjunctive closure of ensures .

Below is how we define Hoare triples, unless , ensures , and $\circlearrowright$ in HOL.

HOL-definition 4.2.1

```
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[6]:

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

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

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

[6]Notice that in the hand definition ${}_P\vdash p\ \mathsf{ensures}\ q$ 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`.

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

As an example, a property of the program `MinDist` in Figure 4.1 is, expressed in the hand notation, the following:

$$\vdash \circlearrowleft (\forall a, b : a, b \in V : d.a.b = \delta_{(V,N)}.a.b)$$

where $\delta_{(V,N)}.a.b$ denotes the actual (simple) minimal distance between a and b.

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)))
```

Let $\mathsf{Trans}.R$ means that R is a transitive relation and Ldisj means that R is disjunctive with respect to its left argument. Let TDC be defined as follows:

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

So, $\mathsf{TDC}.R$ is the smallest closure of R which is transitive and left-disjunctive. Let $\mathfrak{ensures}$ is defined as follows:

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

The progress operator $\rightarrowtail$ can also be defined as the TDC of $\mathfrak{ensures}$:

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

The HOL definition is as follows:

HOL-definition 4.2.2

```
|- !r s. r SUBREL s = (!x y. r x y ==> s x y)
|- !r. TRANS r = (!x y z. r x y /\ r y z ==> r x z)
|- !U. LDISJ U =  (!W y. (?x. W x) /\ (!x::W. U x y) ==> U (??x::W. x) y)
|- !U x y. TDC U x y =  (!X. (SUBREL U X) /\ (TRANS X) /\ (LDISJ X) ==> X x y)
|- !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)
```

◀

4.3 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.

$$
\begin{array}{ll}
\textsf{prog} & \textsf{ABP} \\
\textsf{read} & \{\textsf{wire}, \textsf{Sbit}, \textsf{output}, \textsf{Rbit}, \textsf{ack}\} \\
\textsf{write} & \{\textsf{wire}, \textsf{Sbit}, \textsf{output}, \textsf{Rbit}, \textsf{ack}\} \\
\textsf{init} & \textsf{Sbit} \neq \textsf{ack} \wedge \textsf{Rbit} = \textsf{ack} \\
\textsf{assign} & \\
\quad (\textsf{Send}) & \quad \textsf{if Sbit} = \textsf{ack then wire}, \textsf{Sbit} := Exp, \textsf{next.ack} \\
\quad (\textsf{Receive}) & [\!] \ \textsf{output}, \textsf{Rbit} := \textsf{wire}, \textsf{Sbit} \\
\quad (\textsf{ACK}) & [\!] \ \textsf{ack} := \textsf{Rbit}
\end{array}
$$

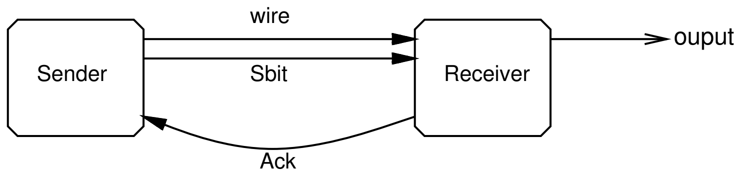

Figure 4.2: Simple Alternating Bit Protocol

Let us consider the program in Figure 4.2. 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:

$$(\forall X :: J \ {}_{\textsf{ALT\_BIT}}\vdash (\textsf{wire} = X) \rightarrowtail (\textsf{output} = X)) \tag{4.3.1}$$

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:

$$
\begin{array}{ll}
& J \vdash (\textsf{wire} = X) \rightarrowtail (\textsf{output} = X) \\
\Leftarrow & \quad \{ \rightarrowtail \textsc{Disjunction} \} \\
& (J \vdash (\textsf{Sbit} = \textsf{Rbit}) \wedge (\textsf{wire} = X) \rightarrowtail (\textsf{output} = X)) \ \wedge \\
& (J \vdash (\textsf{Sbit} \neq \textsf{Rbit}) \wedge (\textsf{wire} = X) \rightarrowtail (\textsf{output} = X))
\end{array}
$$

By the definition of $\rightarrowtail$ the last formula above, and hence also (4.3.1), can be refined to:

$$J \wedge (\textsf{Sbit} = \textsf{Rbit}) \wedge (\textsf{wire} = X) \quad \textsf{ensures} \quad (\textsf{output} = X) \tag{4.3.2}$$

$$J \wedge (\textsf{Sbit} \neq \textsf{Rbit}) \wedge (\textsf{wire} = X) \quad \textsf{ensures} \quad (\textsf{output} = X) \tag{4.3.3}$$

for some invariant J. If we also insist that J is such that Sbit = Rbit implies wire = output then one can prove that (4.3.2) automatically holds.

Let us now see how the derivation above can be done (verified) in HOL.

Code 4.3.1

```
1let AB_PROG_lem = prove(
2   "ENSURES (ALT_BIT Exp Next)
3            (J AND (\s:^XState. ~(s `Sbit` = s `Rbit`) /\ (s `wire` = X)))
4            (\s. s `output` = X)
5    /\
6    ENSURES (ALT_BIT Exp Next)
7            (J AND (\s. (s `Sbit` = s `Rbit`) /\ (s `wire` = X)))
8            (\s. s `output` = X)
9    /\
10   STABLE  (ALT_BIT Exp Next) J
11   ==>
12   REACH   (ALT_BIT Exp Next)  J  (\s. s `wire` = X)  (\s. s `output` = X)",
13  STRIP_TAC THEN SUBST1_TAC lemma
14  THEN MATCH_MP_TAC REACH_SIMPLE_DISJ
15  THEN CONJ_TAC THENL
16  [ MATCH_MP_TAC REACH_ENS_LIFT THEN ASM_REWRITE_TAC[]
17    THEN AB_PROVE_CONF_TAC ;
18    MATCH_MP_TAC REACH_ENS_LIFT THEN ASM_REWRITE_TAC[]
19    THEN AB_PROVE_CONF_TAC ]) ;;
```

◀

The above will prove a theorem stating that (4.3.2) and (4.3.3) together imply (4.3.1). This theorem —actually, hypothesis— is stated in lines 2-12. Lines 2-4 formulate (4.3.3), lines 6-8 formulate (4.3.2), and lines 10-12 formulate (4.3.1). The previous hand derivation is translated into HOL code in lines 13-19 Compare the following with the hand derivation. For example, line 14 applies the $\rightarrowtail$ DISJUNCTION. We obtain two specifications. The decomposition of the $\rightarrowtail$ properties to $\mathsf{ensures}$ properties is done in lines 16 and 19.

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 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:

$$((\mathsf{Sbit} = \mathsf{ack}) \Rightarrow (\mathsf{Sbit} = \mathsf{Rbit})) \ \wedge \ ((\mathsf{Sbit} = \mathsf{Rbit}) \Rightarrow (\mathsf{wire} = \mathsf{output})) \tag{4.3.4}$$

Let us now verify that $\mathsf{ALT\_BIT}$ is a well-formed UNITY program, that J is an invariant, and that (4.3.3) indeed holds. Before we can do that, first we need to define $\mathsf{ALT\_BIT}$ in HOL. This is given below:

HOL-definition 4.3.2

```
|- 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. 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[7]. J1 and J2 are the two conjuncts of the invariant J in (4.3.4).

To prove the well-formedness of a program, we have to check five conditions (see also pages 39). 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[8]. 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.

---

[7]Among other things, UPROG has to convert lists of variables into predicates characterizing the membership of the variables.

[8]The read access of a variable is defined in terms of the $\not\rightarrow$ operato. The operator has a quite complicated definition.

Code 4.3.3

```
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: $(\forall a : a \in \mathbf{a}P : (\mathbf{r}P)^{\mathsf{c}} \not\twoheadrightarrow a)$— 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 4.3.4

```
 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 acti1on  --%
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 (4.3.4), 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 4.3.4.

To prove (4.3.3), that is, $J \wedge (\mathsf{Sbit} \neq \mathsf{Rbit}) \wedge (\mathsf{wire} = X)$ $\mathsf{ensures}$ $(\mathsf{output} = X)$, 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 $\mathsf{unless\ false}$ in Code 4.3.4.

Code 4.3.5

```
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 (4.3.3):

Code 4.3.6

```
 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 (4.3.3). 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[9]. 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:

[9] 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.

```
 1 "HOA
 2  ((J1 AND (J2 AND (\s. ~(s `Sbit` = s `Rbit`) /\ (s `wire`=X)))) AND (NOT(\s. s `output`=X)),
 3  F2R Receive,
 4  (\s. s `output` = X))"
 5    2  ["(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))" ]
 6    1  ["Distinct_Next Next" ]
 7
 8  "PROG(ALT_BIT Exp Next)(F2R Receive)"
 9    2  ["(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))" ]
10    1  ["Distinct_Next Next" ]
11
12 "UNLESS (ALT_BIT Exp Next)
13         (J1 AND (J2 AND (\s. ~(s `Sbit` = s `Rbit`) /\ (s `wire` = X))))
14         (\s. s `output` = X)"
15    2  ["(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))" ]
16    1  ["Distinct_Next Next" ]
17
18  "UNITY(ALT_BIT Exp Next)"
19    2  ["(NOT(L2P AB_Rd)) INVI (F2R(Send Exp Next))" ]
20    1  ["Distinct_Next Next" ]
```

◀

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 4.3.3 and the fact is stated by theorem $\texttt{AB\_UNITY}_{45}$. 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 4.3.5 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.

Bibliography

[And92] Flemming Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 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.

[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.

[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.

[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.

[Dij76] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

[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.

[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.

[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.

[lL77] G. le Lann. Distributed systems —towards a formal approach. In B. Gilchrist, editor, *nformation Processing '77*, pages 155–160. North-Holland, 1977.

[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).

[Pra94] I.S.W.B. Prasetya. Error in the unity substitution rule for subscripted operators. *Formal Aspects of Computing*, 6:466–470, 1994.

[Pra95] I.S.W.B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Dept. of Comp. Science, Utrecht University, 1995.

[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.

[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.

[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.