RSRE
MEMORANDUM No. 4430

ROYAL SIGNALS & RADAR ESTABLISHMENT

WHICH THEOREM PROVER?
(A SURVEY OF FOUR THEOREM PROVERS)

Author: A Smith

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

CONDITIONS OF RELEASE

0088008 BR-115807

********** DRIC U

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4430

Title: WHICH Theorem Prover?
(A Survey of Four Theorem Provers)

Author: A. Smith

Date: October 1990

ABSTRACT

When using Formal Methods to produce verified software, mathematical theorems arise which need to be proved. This memorandum contains the experiences gained in using four theorem provers to prove such theorems. From this experience, a number of recommendations are made on what constitutes a good theorem prover.

THIS PAGE IS LEFT BLANK INTENTIONALLY

1 Introduction

When using Formal Methods to produce verified software, mathematical theorems arise which need to be proved. There are two types of theorems which occur: shallow theorems which mainly arise during program analysis or refinement; and deeper theorems which mainly arise when reasoning about a specification. The former tend to be tedious symbol manipulations, whereas the latter can be more involved. This paper compares the proof of a relatively deep theorem, the condense theorem, using the theorem provers: Edinburgh IPE [1, 2, 3], m-NEVER [4] (from the m-EVES verification system), HOL [5, 6] and the B tool [7]. The main purpose of this paper is to report on the experiences gained in using a number of theorem provers, and from this to make recommendations on what constitutes a good theorem prover. For example, what is a good proof style or user interface?

In addition, the paper shows the results of using m-NEVER with MALPAS[8, 9]. MALPAS is a tool for the analysis of programs, and consists of a suite of analysers, two of which are the semantic analyser and the compliance analyser. The semantic analyser identifies each path through a program and outputs, for each path, a predicate-action pair. The predicate describes the input values to the program that cause it to execute that path, and the fully automatic algebraic simplifier is used to simplify this predicate. The action is a relationship between the input and output values of the program if this particular path is taken. The compliance analyser checks to see whether a program meets its specification. It produces a predicate which describes those input values for which it does not, and the algebraic simplifier is used to simplify this predicate. Thus the predicate *FALSE* means that a program meets its specification. The predicates that MALPAS presents to the user can sometimes be further simplified, sometimes to *FALSE*. Thus a program could meet its specification, but the user could be excused for thinking it did not. Thus it is important to simplify the predicates as far as possible. This was the reason the m-NEVER/MALPAS work was carried out, and is an example of shallow theorem proving. A characteristic of these verification conditions, is their dependence on integer arithmetic, a result of their production from actual software. m-NEVER contains a lot of integer simplification routines, and is particularly suitable for this work. Without these integer simplification routines, the other three theorem provers were not suitable for the MALPAS work.

The structure of the paper is as follows. Section 2 contains an overview of the condense theorem to enable the reader to obtain some intuition into the theorem, followed by a rigorous but informal pen and paper proof. Section 3 describes the proof of the condense theorem using IPE, containing an overview of IPE, how the theorem was tailored to suit IPE, and the experiences and characteristics of the proof. Sections 4, 5 and 6 are the corresponding sections for m-NEVER, HOL and B. Section 7 contains the results of the m-NEVER/MALPAS work. Section 8 compares the four theorem provers, and section 9 contains the conclusions of the whole paper. Appendices A - D show the theory that was added to each of the theorem provers for the condense proof, and appendix E describes the m-NEVER commands used in this paper.

Throughout the paper, the syntax of each theorem prover has not been adhered to, although some of the original syntax has been preserved where necessary. This is to make it easier for the reader to understand, rather than being confused by irrelevant syntax which can be obtained from the user manuals. For example, a common form for the predicates involved has been used.

2 The condense theorem

2.1 Overview

The condense theorem was required for the proof of separability given in [10, 11]. A shared computer system is separable, if the behaviour perceived by each user of the system, is indistinguishable from that which could be provided by an unshared machine dedicated to his private use. The theorem concerns lists, and what follows is an overview of the theorem. The theorem holds for any type of list, but to gain an understanding of the theorem, consider lists of natural numbers, for example $l = [4,3,3,3,7,7,3,9,9]$. The **condense** of l is the new list $[4,3,7,3,9]$, in which any sublist consisting of a repeated element, such as $[3,3,3]$, has been reduced to just that element. Other examples are

$$l = [8,8,7,8,8,8,8,2,2,2] \qquad con(l) = [8,7,8,2]$$

$$l = [5] \qquad con(l) = [5]$$

$$l = [6,6,6] \qquad con(l) = [6]$$

where the function *con* finds the condense of l. A list can also be **partially condensed**. For example if $l = [2,2,2,5,5]$ then a partial condensation is the list $[2,2,5,5]$, that is, the sublist $[2,2,2]$ has been reduced to the sublist $[2,2]$. Another partial condensation of l is $[2,2,2,5]$. The function *pc* gives all the partial condensations of a list, which of course will include the list itself, and the fully condensed list $con(l)$. For example, with $l = [2,2,2,5,5]$ as above, then

$$pc(l) = \{[2,2,2,5,5], [2,2,2,5], [2,2,5,5], [2,5,5], [2,2,5], [2,5]\}$$

The condense theorem is

$$\forall l\ \forall m\ (m \in pc(l) \Rightarrow con\ m = con\ l)$$

This says that the condense of a partial condensation is equal to the condense of the original list. This may seem intuitively obvious, but to prove it formally, axioms capturing the meaning of *con* and *pc* have to be written, and then every step of the proof carried out.

2.2 Informal Proof

First the functions *con* and *pc* must be axiomatised. *con* is easy:

$$con\ [x] = [x] \qquad (1)$$

$$x = head(l) \Rightarrow con(x:l) = con(l) \qquad (2)$$

$$x \neq head(l) \Rightarrow con(x:l) = x:con(l) \qquad (3)$$

where *head(l)* is the first element of a list, for example $head\ [6,2,3,3] = 6$. Also $x:l$ is the list formed by adding the element x to the front of the list l, for example $3:[1,1,2] = [3,1,1,2]$. Axioms 1 - 3 are a recursive definition of *con*. Axiom 1 is the base case, and says that condensing a singleton list leaves it unchanged. Axioms 2 and 3 together form the step case. Axiom 2 says that if the first two elements of a list are identical, then this list can be condensed by throwing away the first element and condensing the rest. Axiom 3 says that if the first two elements are different, then the first element must be kept and the rest condensed. It is instructive to use these axioms to actually condense a list. What follows is the computation of $con[2,2,2,5,1,1]$.

	$con[2,2,2,5,1,1]$	
=	$con[2,2,5,1,1]$	by 2
=	$con[2,5,1,1]$	by 2
=	$2:con[5,1,1]$	by 3
=	$2:5:con[1,1]$	by 3
=	$2:5:con[1]$	by 2
=	$2:5:[1]$	by 1
=	$[2,5,1]$	by definition of ":"

The axiomatisation of *pc* is in the same style but is a little more tricky.

$$pc\ [x] = \{[x]\} \qquad (4)$$

$$x = head(l) \Rightarrow pc(x:l) = x::pc(l) \cup pc(l) \qquad (5)$$

$$x \neq head(l) \Rightarrow pc(x:l) = x::pc(l) \qquad (6)$$

where "$::$" is distributed "$:$". This means that $x::S$, where S is a set of lists, is the new set of lists formed by adding x to the front of every list in S, for example $5::\{[1,1], [7]\} = \{[5,1,1], [5,7]\}$. Axiom 4 is the base case, and says that there is only one partial condensation of a singleton list; namely itself. Axioms 5 and 6 together form the step case. Axiom 5 says that if the first two elements of a list are identical, then this list can be partially condensed by keeping the first element, or by throwing it away, and then partially condensing the rest. Axiom 6 says that if the first two elements are different then the first element must be kept and the rest partially condensed.

Again it is instructive to use these axioms to compute all the partial condensations of a list. What follows is a computation of $pc[6,6,1,1,1]$.

$pc[1] = \{[1]\}$ by 4

$pc[1,1] = 1::pc[1] \cup pc[1]$ by 5
$= \{[1,1]\} \cup \{[1]\}$
$= \{[1,1], [1]\}$

$pc[1,1,1] = 1::pc[1,1] \cup pc[1,1]$ by 5
$= \{[1,1,1], [1,1]\} \cup \{[1,1], [1]\}$
$= \{[1,1,1], [1,1], [1]\}$

$pc[6,1,1,1] = 6::pc[1,1,1]$ by 6
$= \{[6,1,1,1], [6,1,1], [6,1]\}$

$pc[6,6,1,1,1] = 6::pc[6,1,1,1] \cup pc[6,1,1,1]$ by 5
$= \{[6,6,1,1,1], [6,6,1,1], [6,6,1]\} \cup \{[6,1,1,1], [6,1,1], [6,1]\}$
$= \{[6,6,1,1,1], [6,6,1,1], [6,6,1], [6,1,1,1], [6,1,1], [6,1]\}$

With axioms 1 - 6, an informal proof of the condense theorem can now be carried out. The six axioms lend themselves to proof by induction on lists. To prove the theorem, the lemma

$$\forall l\ \forall m\ (m \in pc(l) \Rightarrow head\ m = head\ l)$$

must be proved. That is, the first element of any partial condensation is the same as the first element of the original list. The proof below is by induction on l.

BASE CASE($l = [x]$) Prove $\forall m\ (m \in pc[x] \Rightarrow head\ m = head\ [x])$

Let m be any list such that $m \in pc[x]$. By axiom 4, $pc[x] = \{[x]\}$ and so $m \in \{[x]\}$. If m is in a set with only one element in it, then m must be equal to that element. Thus $m = [x]$ and so $head\ m = head\ [x]$.

STEP CASE From the induction hypothesis $\forall m\ (m \in pc(l) \Rightarrow head\ m = head\ l)$, prove $\forall m\ (m \in pc(x:l) \Rightarrow head\ m = head(x:l))$

Let m be any list such that $m \in pc(x:l)$. Looking at axioms 5 and 6, $pc(x:l)$ depends on whether or not x is equal to the *head* of l. So there are two cases to consider.

Case 1 $x = head\ l$. From axiom 5, $pc(x:l) = x::pc(l) \cup pc(l)$ and so $m \in x::pc(l) \cup pc(l)$. Thus $m \in x::pc(l)$ or $m \in pc(l)$ and so there are two further cases to consider.

Case 1.1 $m \in x::pc(l)$. By the definition of "::", $m = x:n$ for some $n \in pc(l)$. Thus

$head\ m$	$= head(x:n)$	since $m = x:n$
	$= x$	by definitions of *head* and ":"
	$= head(x:l)$	by definitions of *head* and ":"

Case 1.2 $m \in pc(l)$.

$head\ m$	$= head\ l$	by the induction hypothesis as $m \in pc(l)$
	$= x$	by case hypothesis ($x = head\ l$)
	$= head(x:l)$	by definitions of *head* and ":"

Case 2 $x \neq head\ l$. From axiom 6, $pc(x:l) = x::pc(l)$ and so $m \in x::pc(l)$. The argument is then identical to case 1.1 above.

Q.E.D

The proof of the theorem $\forall l\ \forall m\ (m \in pc(l) \Rightarrow con\ m = con\ l)$ is also by induction on l, and follows a similar pattern to the proof of the lemma.

BASE CASE($l = [x]$) Prove $\forall m\ (m \in pc[x] \Rightarrow con\ m = con\ [x])$

Let m be any list such that $m \in pc[x]$. By axiom 4, $pc[x] = \{[x]\}$ and so $m \in \{[x]\}$. Thus $m = [x]$ and so $con\ m = con\ [x]$.

STEP CASE From the induction hypothesis $\forall m\ (m \in pc(l) \Rightarrow con\ m = con\ l)$, prove $\forall m\ (m \in pc(x:l) \Rightarrow con\ m = con(x:l))$

Let m be any list such that $m \in pc(x:l)$. Looking at axioms 5 and 6, $pc(x:l)$ depends on whether or not x is equal to the *head* of l. So there are two cases to consider.

Case 1 $x = head\ l$. From axiom 5, $pc(x:l) = x::pc(l) \cup pc(l)$ and so $m \in x::pc(l) \cup pc(l)$. Thus $m \in x::pc(l)$ or $m \in pc(l)$ and so there are two further cases to consider.

Case 1.1 $m \in x::pc(l)$. By the definition of "::", $m = x:n$ for some $n \in pc(l)$. As $n \in pc(l)$ then by the lemma $head\ n = head\ l$. But as $x = head\ l$ by case hypothesis, then $x = head\ n$. Thus

$con\ m$	$= con(x:n)$	since $m = x:n$
	$= con\ n$	by axiom 2 since $x = head\ n$
	$= con\ l$	by the induction hypothesis as $n \in pc(l)$
	$= con(x:l)$	by axiom 2, since $x = head\ l$ by case hypothesis

Case 1.2 $m \in pc(l)$.

$con\ m$	$= con\ l$	by the induction hypothesis as $m \in pc(l)$
	$= con(x:l)$	by axiom 2, since $x = head\ l$ by case hypothesis

Case $2\ x \neq head\ l$. From axiom 6, $pc(x:l) = x::pc(l)$ and so $m \in x::pc(l)$. By the definition of "::" then $m = x:n$ for some $n \in pc(l)$. As $n \in pc(l)$ then by the lemma $head\ n = head\ l$. But as $x \neq head\ l$ by case hypothesis then $x \neq head\ n$. Thus

$con\ m$	$= con(x:n)$	since $m = x:n$
	$= x:con(n)$	by axiom 3 since $x \neq head\ n$
	$= x:con(l)$	since $con\ n = con\ l$ by the induction hypothesis as $n \in pc(l)$
	$= con(x:l)$	by axiom 3, since $x \neq head\ l$ by case hypothesis

Q.E.D

3 The proof of the theorem with IPE

3.1 Overview of IPE

The version of IPE used was 6.1. The underlying logic of IPE is intuitionistic first order logic. This is basically the same as classical first order logic, but with the law of the excluded middle, namely $A \vee \neg A$ (something is either true or false), missing. The logic is implemented using a Gentzen style sequent calculus, so any proof using IPE is by **subgoaling**. This means that the theorem to be proved, called the **goal**, is broken up into a number of smaller **subgoals**, where a goal or subgoal is a list of hypotheses together with a conclusion. The goal or subgoal is true if the conclusion is a consequence of the hypotheses. So a proof in IPE is carried out in a backward, or top down style. In fact a proof tree is constructed, with a subgoal at each node. There is an **automove** mode, in which the user automatically gets taken to the next unproven subgoal.

Theories can be built up, and some are built into the system, including "Classical", "Equality", "List" and "Set" which are used in the proof of the condense theorem. As its name suggests, the level of interaction by the user is very high, with the user having to do most the steps in a proof. The subgoaling steps involving the underlying logic of IPE are carried out by placing the cursor on the subgoal, and then "clicking" the middle button of the mouse. A new part of the tree representing the new subgoals is then automatically constructed. IPE has an **autoprove** mode which will attempt to do a few subgoaling steps on its own. IPE also supports the use of unconditional rewrite rules. All functions and variables used in IPE are untyped.

3.2 The formalisation of the theorem in IPE

The built-in theories "List", "Set", "Classical" and "Equality" were used. The theory of lists, "List", was built up from the empty list [], and so the built-in list induction used [] as its base case. But recall that axioms 1 - 6 from section 2.2 start from the singleton list [x] as the base case. So the theorem became $\forall l\ \forall m\ (l \neq [] \wedge m \in pc(l) \Rightarrow con\ m = con\ l)$ and the lemma became $\forall l\ \forall m\ (l \neq [] \wedge m \in pc(l) \Rightarrow head\ m = head\ l)$.

The theories "List" and "Set" were quite small and extra axioms had to be added. For example, set union, $\cup$, was declared in the theory "Set", but the axiom $l \in (S \cup T) \Rightarrow (l \in S) \vee (l \in T)$ was not present and so had to be added. It could not be proved from the other axioms of the theory "Set", because the theory did not contain any axioms for $\cup$ at all. Appendix A shows all the functions, axioms and rewrite rules that had to be added. From the informal proof, it can be seen that every so often, the proof splits into cases, for example on whether $x = head\ l$ or $x \neq head\ l$. This meant that that the axiom $A \vee \neg A$ was required, which is part of the theory "Classical". Finally the theory "Equality" was required to deduce that if $e1 = e2$, where $e1$ and $e2$ are expressions, and $P(e1)$ holds, where P is some predicate, then $P(e2)$ holds. For example, if $pc(x:l) = x::pc(l)$ and $m \in pc(x:l)$ then $m \in x::pc(l)$.

3.3 Experiences and characteristics of the proof in IPE

As explained above, the predicate $l \neq []$ had to be added to the theorem and the lemma. From appendix A it can be seen that this predicate also had to be added to the axioms for *con* and *pc* in the step case. Consider the new step case axioms for *con*

$$l \neq [] \wedge x = head\ l \Rightarrow con(x:l) = con(l)$$
$$l \neq [] \wedge x \neq head\ l \Rightarrow con(x:l) = x:con(l)$$

where l and x are generic variables, and can thus be used with any particular values of l and x. When proving the theorem, the expression $con(x:n)$ appeared, where $n \in pc(l)$. But in order to use the above two axioms, to write $con(x:n)$ as one of $con(n)$ or $x:con(n)$, the fact $n \neq []$ had to be known. Thus it soon became apparant that an extra lemma had to be proved, namely $\forall l\ \forall m\ (l \neq [] \wedge m \in pc(l) \Rightarrow m \neq [])$.

With the autoprove mode switched off, the user had to do all the steps in the proof. When the autoprove mode was switched on, the only steps IPE seemed to do for itself were inferences from its basic logic (intuitionistic first order logic as described in section 3.1). That is, it would do some subgoaling steps involving $\wedge$ and $\forall$ for example, automatically, but would not use any of the axioms from "List", "Set", "Classical", "Equality", or any of the added axioms (appendix A). Autoprove actually proved to be a hindrance at times, because for example, it took the lemmas and theorem beyond the point where list induction was appropriate. Thus backtracking was required.

4 The proof of the theorem with m-NEVER

4.1 Overview of m-NEVER

m-EVES is a program verification system, which allows the user to input a program and its specification, and then generates the theorem that must be proved if the program is to meet its specification. m-NEVER is the associated theorem prover which allows the user to try and prove this theorem. m-EVES version 4 was used for the condense theorem. m-NEVER also produces a theorem each time the user defines a recursive function. Proof of this theorem ensures that the function will always terminate when used. Also, each time the user inputs an **axiom** (this is a keyword in m-EVES, and includes statements that are unprovable truths and statements that can be proved from other axioms), m-EVES prompts the user to start the proof of that axiom, although the proof need not be carried out, or can be deferred.

m-NEVER has a mixture of basic and powerful commands, with the powerful commands doing a lot of basic commands in one go. These more powerful commands mean that m-NEVER has a lot of automatic capability. Appendix E gives a description of the m-NEVER commands used in this report.

m-NEVER's proof style is such that, at any time, the user has just one expression, initially the theorem to be proved, which must be simplified to the expression *TRUE*. No matter what commands the user invokes, there is always just one expression. All functions and variables used in m-NEVER must have a type.

4.2 The formalisation of the theorem in m-NEVER

The theorem was proved in m-NEVER without using any built-in theories. This was because although there was a built-in theory of lists, it used the empty list [], and it was decided that the problem this caused in IPE (having to prove an extra lemma) should not be repeated in m-NEVER. Thus a new type representing non-empty lists was declared, together with the functions that act on this type and the relevant axioms that these functions obey. Also there was no built-in theory of sets, and so some set theory had to be added. Appendix B shows the functions and axioms that were added.

From this appendix it can be seen that the functions *con* and *pc* were given a function body, instead of giving their meaning in axioms, as in IPE. This was so that the built-in induction of m-NEVER could be used to prove the lemma and the theorem (Appendix E gives an example of a proof by induction with m-NEVER). As a consequence the *tail* of a list had to be introduced into the function body (the *tail* of a list gives back the old list with everything but its head element, for example *tail*[*1,3,2*] = [*3,2*]). This was because the formal parameter of a function definition (the *l* in the expression *con(l)* = *if...endif* for example) must be a simple variable. Thus *x:l* is not allowed, as this is a constructed term. Thus in IPE, *con(x:l)* was defined in terms of *con(l)*, but in m-NEVER, *con(l)* was defined in terms of *con(tail(l))*.

There are three types of **axioms** in m-NEVER, **facts**, **forward rules** and **rewrite rules**, and each type was used for the condense proof, as can be seen from appendix B. If E is the expression being simplified, then both facts and forward rules are brought in by forming the new expression $F \Rightarrow E$, where F is the fact or forward rule. The difference is that forward rules can brought in automatically by m-NEVER during some of its more powerful commands. When the user defines a forward rule, he also defines an associated **trigger expression**. When m-NEVER spots an instance of the trigger expression, in the expression being simplified, it automatically brings in the forward rule.

For example, suppose the expression being simplified is $length(l) \geq 5$, and there is the ***fact*** $length(l) \geq 9$. Using the command *reduce* on the expression $length(l) \geq 5$ will produce no change. On the other hand the **forward rule** $length(l) \geq 9$, with **trigger expression** $length(l)$, then *reduce* will see the trigger expression in the expression being simplified, bring in the forward rule to produce the expression $length(l) \geq 9 \Rightarrow length(l) \geq 5$, and then simplify it to *TRUE*.

Like forward rules, rewrite rules also enhance m-NEVER's automatic capability, and are again used automatically during some of the more powerful commands. Rewrite rules are used from left to right; when m-NEVER sees an instance of the left hand side of the rule in the expression being simplified, it replaces it by the corresponding instance of the right hand side. Rewrite rules can also contain a guard, which must be satisfied before the rule can be used.

4.3 Experiences and characteristics of the proof in m-NEVER

Whenever a recursive function is declared using a function body, m-NEVER produces a proof obligation. Proof of this will ensure that the function will terminate whenever it is used. Thus m-NEVER produced a proof obligation for both *con* and *pc*, and both of these proof obligations were in fact identical. The proof obligation was simplified to *TRUE* using the first two facts in appendix B.

As already mentioned, the user is always working with just one expression. Even when a proof by induction is requested by the user, m-NEVER will produce just one new expression containing both the base and step case. Working with just one expression meant that, at times, the intellectual grasp of the theorem was lost. This was because it was difficult to see what part of the expression was concerned with what. Also m-NEVER sometimes presents the user with all or just part of the expression in the conditional form *if_then_else_endif*, with subexpressions of this conditional form also sometimes in conditional form. This mixture of forms, and the fact that the user starts off by writing the original expression as a first order predicate, and so is a bit surprised at the form changing, makes such an expression confusing. Also m-NEVER uses a prefix form for its predicates which makes them hard to read. For example the predicate $(P \wedge Q) \Rightarrow (R \vee S)$ is presented as:

IMPLIES(AND(P,Q), OR(R,S))

Had it been possible to split the expression up, it would have been easier to see what each part was concerned with. It seemed to be a case of spotting subexpressions that had something in common with a function or axiom, and then using the function or axiom, hoping it would lead to a simpler expression after simplification. Having said this, due to the more automatic nature of m-NEVER, the last command used in the proof of the theorem (namely *simplify*), simplified an expression occupying about two thirds of a screen, to the expression *TRUE*.

During the proofs, an m-NEVER command could make the expression very large, sometimes several screenfulls; in such cases an intellectual grasp of the proof had definitely been lost, and it was necessary to backtrack. It was much easier to keep the expression to a reasonable size by removing as many of the quantifiers from the expression as possible. This could be done by using the command *prenex*, followed by *open* (see appendix E).

One of the conditions for m-NEVER to carry out a proof by induction, is that the expression must have had all its outermost universal quantifiers removed. Thus the lemma was proved by starting with the expression $m \in pc(l) \Rightarrow head\ m = head\ l$, that is without the $\forall l\ \forall m$ on the front. Alternatively, starting from the expression with $\forall l\ \forall m$ on the front, the *open* command could have been used. Appendix E shows an example of proof by induction using m-NEVER. The command *induct on pc(l)* meant that the expression $m \in pc(tail(l)) \Rightarrow head(m) = head(tail(l))$ became the induction hypothesis, since $pc(l)$ is defined in terms of $pc(tail(l))$. Once the lemma had been proved, it could still be used for any l and m, since they were general variables. Unfortunately, doing a similar thing for the theorem meant that the expression $m \in pc(tail(l)) \Rightarrow con\ m = con(tail(l))$ became the induction hypothesis, and this is not a strong enough induction hypothesis to prove the theorem. It is not strong enough because it has to be used for a list other than m. The hypothesis $\forall m\ (m \in pc(tail(l)) \Rightarrow con\ m = con(tail(l)))$ is what is needed, which is the hypothesis used for all the other theorem provers. Given that the theorem is of the form $\forall l\ \forall m\ P(l,m)$, then the induction hypothesis $\forall m\ P(l,m)$ is the correct hypothesis. When proving the lemma, a weaker induction hypothesis, that is something of the form $P(l,m)$, is sufficient.

To solve this problem the function *pred* (see appendix B) was introduced, which in effect, bracketed the required hypothesis. The theorem could then be proved by starting with the expression *pred(l)*. As this expression does not have any universal quantifiers at the front, m-NEVER will allow induction. The command *induct on pc(l)* (or *induct on con(l)*) then produces the expression

$$[\neg(A \vee B) \Rightarrow pred(l)] \wedge$$
$$[(A \wedge pred(tail(l))) \Rightarrow pred(l)] \wedge$$
$$[(B \wedge pred(tail(l))) \Rightarrow pred(l)] \wedge$$

where

A is the expression $length(l) \neq 1 \wedge head(l) = head(tail(l))$ and
B is the expression $length(l) \neq 1 \wedge head(l) \neq head(tail(l))$

The actual definition of *pred(l)* can then be invoked. This technique of hiding the universal quantifier thus produces the required induction hypothesis. This seems to be an area where m-NEVER could be improved, by allowing induction on an expression containing outermost universal quantifiers. While writing this paper, it has been learnt that there is some current work going on which is addressing this problem.

5 The proof of the theorem with HOL

5.1 Overview of HOL

The version of HOL used was 1.11. The HOL theorem prover is an implementation of Higher Order Logic. Theorems can be proved using the HOL theorem prover in either a forward or backward (subgoaling) style. **Inference rules** which take theorems as arguments and give new theorems as results, are used for forward proof, while **tactics** are used for backwards proof. A tactic takes a goal (which at the start will be the theorem to be proved) and returns a number of smaller subgoals. Tactics are then applied to these subgoals and so on. The condense theorem was proved in the backward style. The user interacts with HOL via the functional programming language ML. Tactics can be composed together using **tacticals** to produce new tactics with more power. HOL comes with a lot of built-in inference rules, tactics and tacticals, but new ones can be written by the user in ML. HOL also comes with a number of built-in theories, including lists and sets which are used in the proof of the condense theorem. New theories can be added by the user. Both the HOL logic and ML are typed.

5.2 The formalisation of the theorem in HOL

HOL has comprehensive built-in theories of lists and sets which contained everything required for the condense proof, apart from axioms for the functions *con*, *pc* and "::". Also HOL has a built-in list induction tactic. Unfortunately all the built-in theories and the list induction tactic use the empty list []. Recall that from the IPE proof, this meant that an extra lemma had to be proved. In this case it was less work to use the built-in theories and list induction tactic (as very little had to be added), and prove the extra lemma, rather than build up new theories, and write a new list induction tactic to save proving the extra lemma. Appendix C shows how only axioms defining the functions *con*, *pc* and "::" had to be added.

5.3 Experiences and characteristics of the proof in HOL

As mentioned above, working with the empty list meant that an extra lemma had to be proved; namely $\forall l\ \forall m\ (l \neq [] \wedge m \in pc(l) \Rightarrow m \neq [])$. The first lemma, namely $\forall l\ \forall m\ (l \neq [] \wedge m \in pc(l) \Rightarrow head\ m = head\ l)$ was proved in HOL by using a tactic, HOL coming back with subgoals, and the user then applying new tactics to these subgoals, and so on. Because the proof of both lemmas and the theorem were known to be similar, a compound tactic was formed from the individual tactics of the interactive proof just performed. The individual tactics were composed together using tacticals. By making this tactic slightly more complicated than was required to prove the first lemma, it was found that it would prove both lemmas. This tactic is shown in appendix C. It was then used on the theorem, with HOL returning three subgoals, which were then proved using other tactics together with the two lemmas. So not only did this tactic prove both lemmas, but it also went a long way to proving the theorem.

6 The proof of the theorem with B

6.1 Overview of B

The B tool is a general tool for rewriting expressions using **rules** contained within rule files. Thus it has a number of applications, one of which is theorem proving. Another application might be to have a rule file containing rules which convert a program written in a high level language to machine code and thus simulate a compiler. This report is only concerned with the B tool in its theorem proving capacity. B also uses a subgoaling style but only has a handful of built-in tactics. Others are added by the user as rules. All functions and variables input by the user are untyped.

6.2 The formalisation of the theorem in B

As mentioned above, very little is built in to B, with the user adding **rules** as required. For example, both rewrite rules and new user defined tactics are input as rules. Appendix D shows the rules that were added for the condense proof. Rules are collected together in **theories** as shown. For example, consider the theory *con*. This contains the three rules, the first of which is *con* $[x] = [x]$. This will be used by B as a rewrite rule, that is from left to right, replacing any instance of *con* $[x]$ that it finds in the conclusion of the goal, with $[x]$. Another example is the theory *equality2* which contains the rule $x = y \Rightarrow z{:}x = z{:}y$. This is an example of a user defined tactic. When B sees an instance of $z{:}x = z{:}y$ in the conclusion of a goal, it will replace the conclusion with $x = y$.

So rewrite rules have the form $C = D$, and new tactics have the form $C \Rightarrow D$. With new tactics, the LHS (the C) can also be a conjunction of two or more expressions. For example, consider the theory *listinduct* from appendix D, which contains a rule of the form *(base case)* $\wedge$ *(step case)* $\Rightarrow \forall l\, P$. This rule is a tactic to carry out list induction, and from a goal with conclusion $\forall l\, P$, produces two subgoals, namely the base case and the step case. Rewrite rules and tactics are used by B in the following way. The user defines a tactic, composed of theories and tacticals, for example

prv; listinduct; (con; pc; cases; hd; undisch; distcons; equality2)~

which appears in the theory *prove* in appendix D. The tacticals ";" and "~" are similar to THEN and REPEAT in HOL. B moves through the tactic from left to right, looking in the named theories for rewrite rules or new tactics, and then applying them as described above.

Rules of the form $C \Rightarrow D$, are also used in a forward direction, that is from left to right, and B does this by using a **forward tactic**, an example of which is

(results; con; pc; singleset; equality1)~

which again appears in the theory *prove* in appendix D. It is used in the following way. First the built-in tactic DED is used to convert a subgoal of the form $H \vdash E1 \Rightarrow E2$ to a new subgoal of the form $[H, E1] \vdash E2$. B then automatically moves through the forward tactic from left to right, trying to match $E1$ to the antecedent (the C) of a rule of the form $C \Rightarrow D$. If successful D is then added to the hypotheses. B then carries on looking through the forward tactic trying to match either $E1$ or D to the antecedent of a rule. The forward tactic has similarities with the tactic IMP_RES_TAC in HOL.

Thus the tactic acts on the conclusion of a goal, while the forward tactic acts on the hypotheses of the goal. The name "forward tactic" is consistent with the idea of going forwards in the proof, from the hypotheses, while the tactic is concerned with backwards proof, splitting a goal into a number of subgoals. The tactic and forward tactic shown above, from the theory *prove*, were used to prove both the lemma and theorem.

Both the lemma and the theorem were proved in B's interactive mode, which meant that B would give the user the option of refusing a rule it had found applicable. It also meant that the user could use built-in tactics, such as DED, during B's traversal of the tactic. The theory *prv* contains the line *results+g*. This meant that after B had proved the lemma, it was added to the theory *results*, which was empty during the proof of the lemma, so that it could be used during the proof of the theorem.

B will only match rules to expressions if the rules contain single letters as the generic parameters. These single letters (either lower or upper case) are known in B as **jokers**. Also expressions of the form $[l := E]P$ in the theory *listinduct* mean replace all free occurrences of l in P with the expression E. A condition that must be satisfied for B to perform this substitution is that l must be a *variable* (basically a string of at least two characters), or that the expression *vrb(..., l, ...)* appears in the hypotheses of the goal at the time B tries to perform the substitution. For this reason the expression *vrb(l, m)* appears in the lemma and theorem to be proved, so that by the time B tries to perform the substitutions as described above, *vrb(l, m)* will appear in the hypotheses.

Rules can also have *guards*. See for example the two rules in the theory *cases*. Both have a guard of the form *inhyp(E)*, where E is an expression. B will only use the rule if it is matched to the conclusion of the current subgoal, and E appears in the hypotheses of that subgoal (hence the name inhyp).

6.3 Experiences and characteristics of the proof in B

As mentioned above, the proofs were carried out in B's interactive mode. This meant that the built-in tactics, for example DED, could be used when required. While B goes through the tactic, the user can invoke a built-in tactic and B will then carry on with the tactic and so on. Given time, the proof could have been made automatic, by, amongst other things, putting all the required built-in tactics in the right places in the tactic. Also, futher use of guards (*inhyp* etc) and splitting up the theories may be required. The use, for example, of *inhyp* in this situation is useful to automatically refuse the use of a rule that was found to be applicable, since it might be known that use of that rule at that time was not wanted. Also splitting up the theories into smaller theories is useful to remove rules from consideration at a certain point in the tactic.

Rules are just typed in, that is the functions that appear in the rules do not have to be declared first. This meant that appendix D was just typed straight in, without having to declare any of the functions used first.

The built-in tactic HYP was found to be useful during the proofs. This tactic treats the hypotheses as just another theory. On a number of occasions, a goal of the form $[C, C \Rightarrow D] \vdash D$ had to be discharged. Using HYP, the hypotheses of the goal, namely $[C, C \Rightarrow D]$ are treated as another theory. Thus B noticed that D was the conclusion of the goal, and that $C \Rightarrow D$ was in the hypotheses, and produced the new subgoal $[C, C \Rightarrow D] \vdash C$ which is immediate since the conclusion C appears as one of the hypotheses. Thus the hypothesis $C \Rightarrow D$ was used as a tactic.

Goals of the form $[C, C \Rightarrow D] \vdash D$ also appeared when performing the condense proof with HOL, and it is interesting to see how they were discharged in HOL. In fact RES_TAC was used, which uses the hypotheses C and $C \Rightarrow D$ to deduce the new hypothesis D, thus generating the new subgoal $[C, C \Rightarrow D, D] \vdash D$. This goal is immediate, since the conclusion D appears as one of the hypotheses.

The second rule in the theory *distcons* is

$$(P \Rightarrow Q)$$
$$\Rightarrow$$
$$((\exists n.P) \Rightarrow Q)$$

which, during the proof is used to replace a goal with conclusion $(\exists n.P) \Rightarrow Q$ with a new goal having conclusion $P \Rightarrow Q$. This is only valid, providing the existentially quantified variable n does not appear free anywhere else in the goal (apart from in P). That is, it must not appear in the conclusion, and it must not appear in the hypotheses. So the above rule should have a guard to ensure this, but it was not clear how to write this in B, and so it was left out. When the rule was used, a manual check was performed by the user, to ensure that n did not appear free elsewhere in the goal. B does have a built-in guard "\" which is used in the form *variable\formula* to ensure that the *variable* does not appear free in the *formula*. But it is not clear how to use this to ensure that a variable does not appear free anywhere in the goal. There is also a way of prompting the user for a new name for n when the rule is used, but this is open to error if the user gives a new name that appears free elsewhere. All this does seem an area where B could be improved, by supplying some simple built-in tactics to deal with goals containing $\exists$. After all, $\exists$ is a built-in symbol, just like $\forall$, and there is a built-in tactic to deal with $\forall$, namely GEN. GEN strips off a $\forall$ and ensures that the universally quantified variable is not free elsewhere, and if it is, renames it.

7 Using m-NEVER to further simplify MALPAS predicates

There are some predicates that MALPAS can not simplify any further. What follows is a selection of some of these predicates, and the results of using the interactive theorem prover m-NEVER to simplify these predicates further. The predicates were simply hand typed into m-NEVER. The particular versions used were MALPAS version 4.3 and m-EVES version 4.

a) The predicate below is the result of compliance analysis of a program which takes three integers a, b and c and decides whether a triangle can be formed with sides a, b and c, and, if so, the type of triangle (right-angled, isoceles etc).

$$(c > 0 \wedge a = b \wedge a * a = b * b + c * c \wedge a = c) \vee$$
$$(c > 0 \wedge a = b \wedge a * a > b * b + c * c \wedge a = c)$$

Using m-NEVER with the command *simplify*, simplified the above expression to *FALSE*.

b) The following propositional formulae

$$p \wedge \neg q \wedge (p \Rightarrow q)$$

and $\quad (p \wedge q) \vee (p \wedge \neg q) \vee (q \wedge \neg p) \vee (\neg p \wedge \neg q)$

where p and q are boolean variables, can be simplified using m-NEVER with the command *simplify* to *FALSE* and *TRUE* respectively.

c) The expression $x^2 = 0 \wedge x \neq 0$ can be simplified to *FALSE* using m-NEVER with the command *simplify*.

d) The predicate below is the result of compliance analysis of a program concerning eight bit addition with protected overflow.

$$(x > -129 \wedge y > -129 \wedge x + y < -127 \wedge y\ PLUS\ x \neq -127 \wedge x < 0 \wedge$$
$$y < 0 \wedge y\ PLUS\ x < 0) \vee (x = 0 \wedge y = -128) \vee (x = -128 \wedge y = 0)$$

where the function *PLUS* was defined in MALPAS using the rewrite rules

$a\ PLUS\ b \longrightarrow a + b$	*if* $-128 \leq a + b \wedge a + b \leq 127$	(rule 1)
$a\ PLUS\ b \longrightarrow a + b - 256$	*if* $a + b > 127$	(rule 2)
$a\ PLUS\ b \longrightarrow a + b + 256$	*if* $a + b < -128$	(rule 3)

with x, y, a and b all declared as integers. MALPAS cannot simplify the expression any further, because it cannot make further use of the rewrite rules to eliminate the two occurrences of *PLUS* left in the expression. This is because it cannot prove any of the three guards of the rewrite rules. Unfortunately using m-NEVER with the command *reduce* (the most powerful command) the rules were not used either, because it too could not prove any of the guards. But notice however, that in the first disjunct of the expression there is the predicate $x + y < -127$, which means that for this disjunct either $x + y = -128$ (and so rule 1 applies) or $x + y < -128$ (and so rule 3 applies). Using the command *split* $x + y = -128$ (which performs a case analysis; $x + y = -128$ is either true or false) on the expression followed by *reduce* gave

$$x + y = -128 \wedge x > -129 \wedge y > -129 \qquad \text{(expression 1)}$$

The *split* meant that *reduce* could then prove the guards of rules 1 and 3, and so use the rules. This is a good example of being able to give the simplifying process a little "push" when required (in this case with the *split*).

e) Consider the same expression as in **d**, but with the function *PLUS* given a body in m-NEVER, that is

if $-128 \leq a + b \wedge a + b \leq 127$ *then* $a + b$
elseif $a + b > 127$ *then* $a + b - 256$
else $a + b + 256$
end if

rather than using rewrite rules to define it. When *reduce* was applied to the expression, the two occurences of *PLUS* were replaced automatically by the above body, instantiated accordingly, and then the whole expression was simplified. This resulted in the expression

$$x > -129 \wedge y > -129 \wedge x + y < -127 \wedge \qquad \text{(expression 2)}$$
$$(x \geq 0 \vee y \geq 0 \vee x + y \geq -128)$$

When using *reduce* any functions such as *PLUS* are invoked automatically and replaced by their corresponding body, instantiated accordingly. *reduce* then simplifies the resulting expression. Thus it is sometimes better to translate rewrite rules in MALPAS which are used to give meaning to functions, to function bodies in m-NEVER since *reduce* will then invoke the functions automatically.

Of course expression 2 is logically equivalent to expression 1 and can be simplified to it using the command *disjunctive* (which puts an expression into disjunctive normal form) to conjoin $x + y < -127$ with $x + y \geq -128$, followed by *split* $x + y = -128$, followed by *simplify*. The *split* is needed to help the subsequent *simplify* command to simplify the conjoin $x + y < -127 \wedge x + y \geq -128$ to $x + y = -128$. This seems to be an area where m-NEVER could be improved so that it could simplify such expressions automatically. Of interest, in this particular example, is that expression 2 can be simplified to expression 1 by leaving out the first step mentioned above (*disjunctive*).

f) Suppose the following two rewrite rules are entered into MALPAS:

$$(l \wedge m) \wedge n \longrightarrow l \wedge (m \wedge n) \qquad \text{(rule 4)}$$
$$last(l \wedge [a]) \longrightarrow a \qquad \text{(rule 5)}$$

where l, m and n are lists, $[a]$ is the singleton list containing the element a, $\wedge$ appends two lists, and *last* gives the final element of a list. MALPAS cannot simplify the expression

$$last(l \wedge (m \wedge [a])) = a \qquad \text{(expression 3)}$$

because it cannot apply either of the above rewrite rules. However if the two rules are translated into rewrite rules in m-NEVER, the command *use* can make available the underlying equation of the rule as an assumption of the current expression being simplified. It constructs an implication consisting of the underlying equation and the current expression (see below for example). This underlying equation can be used in both directions, that is, from left to right and from right to left, unlike the original rewrite rule which can only be used from left to right.

For example, when simplifying expression 3, the command

use rule 4 $l = l, m = m, n = [a]$ gives the new expression

$$(l \wedge m) \wedge [a] = l \wedge (m \wedge [a]) \Rightarrow last(l \wedge (m \wedge [a])) = a$$

Next, *equality substitute* $l \wedge (m \wedge [a])$ gives the expression

$$(l \wedge m) \wedge [a] = l \wedge (m \wedge [a]) \Rightarrow last((l \wedge m) \wedge [a]) = a$$

Next, *use rule 5* $l = l \wedge m, a = a$ gives the expression

$$(l \wedge m) \wedge [a] = l \wedge (m \wedge [a]) \wedge last((l \wedge m) \wedge [a]) = a \Rightarrow last((l \wedge m) \wedge [a]) = a$$

Finally *simplify* yields the expression *TRUE* since the expression *last((l ^ m) ^ [a]) = a* is common to both sides of the implication. Interestingly, in this particular case, expression 3 can be simplified to *TRUE* using the command *use rule 4* followed by *use rule 5* followed by *simplify*. Here the commands *use rule 4* and *use rule 5* make available the underlying equations as assumptions, but this time with the variables universally quantified, as no instantiations were supplied. The command *simplify* finds the particular instantiations required and performs the equality substitution automatically.

8 Comparisons between the four theorem provers

IPE, HOL and B use a subgoaling style of proof, whereas m-NEVER works with just one expression all the time. The idea of subgoaling seemed to be much more natural than to try to simplify one expression. With m-NEVER it was difficult to see what each part of the expression was concerned with. This was especially true if the expression grew quite large, or if part or all of the expression was presented to the user in the conditional *if_then_else_endif* form. This meant that an intellectual grasp of the proof was sometimes lost. Also controlling the size of the expression was at times a problem. Sometimes the expression grew very large and so backtracking was required.

On the question of interaction by the user verses automation, m-NEVER, HOL and B have a lot more automation than IPE. The size of a single subgoaling step in HOL or B can be equivalent to many subgoaling steps in IPE. In fact with IPE every subgoaling step represents a single basic logical step. With **autoprove** off, having to do every such step in the condense proof was very tedious. With autoprove on, IPE carried out some steps on its own, but only at the level of its underlying logic, that is involving $\wedge$ and $\vee$ for example. It did not do any steps involving list or set theory for example. Also it took the user past the stage where list induction was appropriate, and thus backtracking was required. In HOL and B, tactics, which carry out the subgoaling steps, can be composed together with tacticals to form more powerful tactics, thus increasing the level of automation. HOL has a lot of built-in tactics, whereas B has only a few. New tactics are added to B in the form of **rules** by the user, for example a tactic must be added to B to perform a case analysis.

In both HOL and B, the proof can be fully automated by using one big tactic, but, using them both interactively, it was found that, for the condense proof, B needed slightly more interaction than HOL. For example, if B finds a rule that it can use as a rewrite rule it will ask the user to accept it or decline it, and then go on to find another rule and ask the user again. But in HOL, the tactic REWRITE_TAC can use many rewrite rules in one go. Also in HOL, completely new tactics can be written by the user in ML, that is tactics that can not be built up from existing tactics and tacticals. m-NEVER has a mixture of basic and more powerful commands. The final command used during the proof of the condense theorem, namely *simplify*, simplified an expression occupying about two thirds of a screen, to *TRUE*.

IPE has a good user interface, being able to "click" with the mouse on a goal to produce the next subgoal. On the screen, each subgoal and the name of each subgoaling step contains plenty of English. The proof is constructed as a tree, and there is an **automove** mode which will automatically take the user to the next unproven subgoal in the tree. B keeps track of all the subgoals, presenting the user with the next one to be proved. HOL also keeps track of all the subgoals, and the user can change the order in which the subgoals are to be proved.

m-NEVER uses a prefix form for its predicates, which makes them hard to read. IPE, HOL and B use an infix form. Also, as mentioned above, m-NEVER can sometimes present the user with an expression where all or part of it is in the conditional *if_then_else_endif* form. This was found confusing because the expression could be a mixture of forms (first order predicates together with the conditional form). Also the user has input the original expression in the first order predicate form and so is a bit surprised to see this new form. While writing this paper, it has been learnt that there is some current work going on, which is addressing this problem.

Formalising the theorem in each theorem prover was interesting. For example in IPE, HOL and B, the meaning of the functions *con* and *pc* were input as axioms, whereas in m-NEVER the meanings were given in function bodies. If the meanings were input into m-NEVER as axioms, the built-in induction could not be used, since this looks for functions that have been defined recursively using a function body. As a consequence, the *tail* of a list had to be introduced, since the formal parameter of a function in m-NEVER must be a simple variable; in this case it was chosen to be *l* (*x:l* is not allowed, as this is a constructed term). Thus in IPE, HOL and B, *con(x:l)* was defined in terms of *con(l)*, while in m-NEVER, *con(l)* was defined in terms of *con(tail(l))*.

9. Conclusions

The proof of the condense theorem was found to be easiest with HOL. A subgoaling approach seemed more natural than m-NEVER's approach of working with just one expression. However, while writing this paper, it has been learnt that a new system called EVES, with a new theorem prover called NEVER, is being implemented, in which the user will be able to split up the expression into a number of smaller subexpressions. Also, given an expression, the user will be able to direct a prover command to act on just a chosen subexpression. Although IPE used a subgoaling approach, the size of each subgoaling step was too small, with the user having to do every step in the proof. There was not much to choose between HOL and B. Only the bare minimum of axioms had to be added to HOL, whereas quite a few rules had to be added to B, some to do such things as case analysis. When proving the theorem interactively, after the axioms and rules had been added, then slightly more interaction was necessary with B than with HOL. Admittedly an extra lemma had to be proved with HOL, than in B, due to the use of the empty list [] in HOL.

Based on the experience gained from using the four theorem provers, the following properties would seem to be desirable.

(a) A subgoaling approach.

(b) A high degree of automation when required. Also the ability to write completely new proof procedures initially unknown to the system.

(c) A good user interface with menus, containing such things as the unproven subgoals, tactics, tacticals and theorems. The ability to access items from these menus by pointing and clicking with a mouse.

(d) Fonts, to enable the conventional mathematical symbols to appear on the screen. For example, to enable $\forall$ to appear on the screen, instead of *!*.

(e) The ability to insert the theorem prover into a development environment.

(f) The ability to store and replay a proof, that is, to be able to run a proof again automatically, without any user interaction. This is useful for automatic maintenance of a formal development when, say, the specification changes, and proofs of theorems need to be replayed to see if they still hold.

(g) Good documentation.

What follows is an assessment of how each of the four theorem provers used in this paper lives up to the points made above.

IPE uses a subgoaling approach, but each subgoaling step is a single basic logical step. Apart from the autoprove mode, which is limited, there is no way of carrying out large subgoaling steps, which would help to automate the subgoaling process. Also there is no way of writing completely new proof procedures. IPE does have a good user interface, with menus accessed by pointing and clicking with a mouse. Some of the conventional mathematical symbols appear on the screen, for example $\forall$ and $\exists$, but others do not, for example *&* appears instead of $\wedge$. It is not known whether IPE could fit easily into a development environment, but a proof can not be stored and replayed. IPE comes with reasonable documentation.

m-NEVER does not use a subgoaling approach, but instead uses a single expression, which the user must simplify to *TRUE*. m-NEVER does have some very powerful commands, which give it a high degree of automation when required, although there is no way to write completely new proof procedures. The user interface does not contain menus, and conventional mathematical symbols do not appear on the screen. m-NEVER forms part of the m-EVES verification environment. m-NEVER is written in Lisp, and was run by the author on a SUN-3 workstation, under the UNIX operating system. The usual way to interact with m-EVES is through the Emacs editor, although this method was not used for this paper. By interacting through Emacs it is possible to store proof commands in a file and replay them. m-NEVER comes with good documentation, with a clear description of each user command.

HOL uses a subgoaling approach, and has the facility to construct large powerful tactics, which help to automate the subgoaling process. It also has the facility to be able to write completely new tactics, in ML. The user interface does not contain menus, and conventional mathematical symbols do not appear on the screen. HOL is driven from the programming language ML, and so a HOL proof is an ML program. The integration of this into a configuration control system, say, could then follow the standard techniques used with other compilers. HOL comes with very good documentation, although the documentation for some of the built-in tactics and tacticals is incomplete.

B uses a subgoaling approach, and has the facility to construct large powerful tactics, which help to automate the subgoaling process. Completely new tactics are written by the user in the form of rules. B has a reasonable user interface, using menus, although items in these menus are accessed by number rather than by pointing with a mouse. Conventional mathematical symbols do not appear on the screen. B contains a development environment within itself (recall that B is not just a theorem prover, but a general rewriting tool). B can be used to develop software, with proof obligations being produced automatically along the way. It is not known how easily B would fit into another development environment, but a proof can be stored and replayed, by constructing a large tactic and a large forward tactic, composed of all the necessary theory files, built-in tactics and tacticals. At the time the B tool was used for this paper, there was no user manual, although there is a language manual, which descibes the symbols and syntax used by B, together with the meaning of the symbols.

No comparison of theorem provers is complete without a discussion of **soundness**, that is, if a statement is proved by the program, is it in fact a theorem within the logic? There are three main points to consider. Firstly, does the underlying logic of the theorem prover exist, and is it sound? Secondly, is there the possibility that contradictory axioms might be added by the user? Thirdly, is the software that implements the theorem prover correct, that is, are we confident it will not prove a wrong theorem? On this third point, high assurance is gained by having the structure of the software reflect the logic, so that, if possible, the correctness depends only on a trusted kernel of software.

With IPE, the underlying logic is intuitionistic first order logic. This is sound, but other logics are built up from it, by adding axioms. So there is the chance that inconsistencies might be added. It is not known how the software of IPE reflects the logic.

With m-NEVER, the underlying logic is basically first order predicate calculus, which is sound. But again inconsistent axioms can be added, because although m-NEVER prompts the user for a proof of each statement entered, the proof can be deferred. However, it is possible to check on the status of each statement entered by the user, that is, whether it is proven or unproven. The originators of m-EVES encourage the proof of all statements entered "either directly or by constructing models using the library mechanism of m-EVES" [12].

With HOL, the underlying logic is higher order logic, which is sound. Again there is the possibility of introducing contradictory axioms, but HOL provides the ability to use **definitions**. Definitions are guaranteed to be consistent, and HOL provides the user the ability to check that a statement is actually a definition. The ML type structure gives extra assurance of soundness. This is because objects of type "thm" can only be constructed from other objects of this type using the basic axioms and inference rules of the HOL logic. Thus the correctness of the software implementing the HOL theorem prover depends only on the ML typechecker and the implementation of the HOL logic. This is a relatively small amount of software as compared with other theorem provers.

With B, there is very little underlying logic, but what there is has not been written down, and so it is not known whether it is sound. There is the possibility of the user introducing inconsistent axioms, especially as more axioms have to be added to B, as compared with other theorem provers. B is written in PASCAL, but the structure of the software is unknown.

Finally, the use of m-NEVER to further simplify MALPAS outputs proved successful. The fact that m-NEVER had a level of interaction, as well as having powerful automatic commands, was useful in giving the simplification process a little "push" at times. Admittedly m-NEVER was only used on MALPAS **outputs**, that is, already simplified expressions, rather than the original unsimplified expressions, which would have been a lot bigger. This point is made because of the problems encountered in controlling the size of the expression during the condense proof. However, it would be interesting to see how m-NEVER performs with these unsimplified expressions. Certainly m-NEVER looked like a useful theorem prover to use in conjunction with MALPAS, as a means of introducing some capability for interactive proof.

References

1. Ritchie B. "The Design and Implementation of an Interactive Proof Editor", Thesis, CST-57-88 (LFCS-88-68). October 1988.

2. Ritchie B. and Taylor P. "The Interactive Proof Editor, An Experiment in Interactive Theorem", ECS-LFCS-88-61. July 1988.

3. Taylor P. and Jones C. "The Interactive Proof Editor User Guide", LFCS-TN-11. May 1988.

4. m-EVES Collected Papers (Prepared for the Sun-3 m-EVES Version 4 Distribution Tape). Odyssey Research Associates, Inc. September 1989.

5. HOL Reference Manual (Description). SRI International, Cambridge Research Centre (1989)

6. HOL Reference Manual (Tutorial). SRI International, Cambridge Research Centre (1989)

7. Abrial J. R. "B Reference Manual". Draft #6. BP internal document. May 1990

8. B. D. Bramson "Tools for the Specification, Design, Analysis and Verification of software". RSRE report number 87005 (1987)

9. MALPAS User Guide (Version 4.3). Rex, Thompson and partners. February 1990.

10. Rushby J.M. "The design and verification of secure systems", Proc 8th ACM Symposium on Operating System Principles, Asilomar, California, USA, December 1981. (Available as ACM Operating Systems Review, vol 15, no 5).

11. Sennett C.T. and Macdonald R. "Separability and Security Models", RSRE report no: 87020. November 1987.

12. Pase B. and Kromodimoeljo S. Private communication. October 1990.

Acknowledgements

The author would like to thank Bill Pase and Sentot Kromodimoeljo of Odyssey Research Associates Inc, Ib Sorenson and Dave Nielson of BP Research, and Mike Gordon of Cambridge University for their help and comments.

Appendix A

The functions, axioms and rewrite rules added to IPE for the condense proof

functions

con, *pc*, *head*, *::*, {_} ({_} forms the singleton set of an element)

axioms (all variables are generic, that is, can be used with any particular value)

$l = [] \Rightarrow con(x{:}l) = x{:}[]$

$l \neq [] \wedge x = head(l) \Rightarrow con(x{:}l) = con(l)$

$l \neq [] \wedge x \neq head(l) \Rightarrow con(x{:}l) = x{:}con(l)$

$l = [] \Rightarrow pc(x{:}l) = \{x{:}[]\}$

$l \neq [] \wedge x = head(l) \Rightarrow pc(x{:}l) = x{::}pc(l) \cup pc(l)$

$l \neq [] \wedge x \neq head(l) \Rightarrow pc(x{:}l) = x{::}pc(l)$

$l \in x{::}S \Rightarrow \exists m\,(l = x{:}m \wedge m \in S)$

$x \in \{y\} \Rightarrow x = y$

$l \in S \cup T \Rightarrow l \in S \vee l \in T$

rewrite rules (again, variables are generic)

$head(x{:}l) \longrightarrow x$

THIS PAGE IS LEFT BLANK INTENTIONALLY

Appendix B

The functions and axioms added to m-NEVER for the condense proof

functions

head, tail, length, :, {_}, ∪, ::, ∈ (declared with no body)

con (l) = if length(l) = 1 then l
elseif head(l) = head(tail(l)) then con(tail(l))
else head(l):con(tail(l))
end if

pc (l) = if length(l) = 1 then {l}
elseif head(l) = head(tail(l)) then head(l)::pc(tail(l)) ∪ pc(tail(l))
else head(l)::pc(tail(l))
end if

pred (l) = begin ∀m (m ∈ pc(l) ⇒ con(m) = con(l)) end

axioms (all variables are generic, that is can be used with any particular value)

(a) facts

$length(l) \geq 1$
$length(l) \neq 1 \Rightarrow length(tail(l)) \leq length(l)$
$l \in x::S \Rightarrow \exists m\ (l = x:m \wedge m \in S)$

(b) forward rules

$l = x:m \Rightarrow length(l) \neq 1$ (trigger expression $length(l)$)

(c) rewrite rules

$m \in \{l\} \longrightarrow m = l$
$l \in S \cup T \longrightarrow l \in S \vee l \in T$
$head(x:l) \longrightarrow x$
$tail(x:l) \longrightarrow l$

THIS PAGE IS LEFT BLANK INTENTIONALLY

Appendix C

The functions and axioms added to HOL for the condense proof

functions

con, pc, ::

axioms

$\forall x\ (con\ [x] = [x]) \wedge (pc\ [x] = [x]\ INSERT\ \{\})$ A1

$\forall x\ \forall l\ (l \neq [] \wedge x = head\ l \Rightarrow con(x:l) = con\ l \wedge pc(x:l) = x::pc(l) \cup pc(l))$ A2

$\forall x\ \forall l\ (l \neq [] \wedge x \neq head\ l \Rightarrow con(x:l) = x:con\ l \wedge pc(x:l) = x::pc(l))$ A3

$\forall x\ \forall l\ \forall S\ (l \in x::S = \exists m\ (l = x:m \wedge m \in S))$ A4

The tactic used to prove both lemmas, and reduce the theorem to three subgoals

```
LIST_INDUCT_TAC THEN
REWRITE_TAC[NOT_CONS_NIL; HD] THEN
REPEAT GEN_TAC THEN
ASM_CASES_TAC "(l : (*)list) = []" THENL
  [ASM_REWRITE_TAC[A1; IN] THEN
   DISCH_TAC THEN
   ASM_REWRITE_TAC[NOT_CONS_NIL; HD];
   ASM_CASES_TAC "h = HD(l : (*)list)" THENL
     [IMP_RES_TAC A2 THEN
      ASM_REWRITE_TAC[IN_UNION; A4] THEN
      STRIP_TAC THENL
        [RES_TAC;
         ASM_REWRITE_TAC[NOT_CONS_NIL; HD]];
      IMP_RES_TAC A3 THEN
      ASM_REWRITE_TAC[A4] THEN
      STRIP_TAC THEN
      ASM_REWRITE_TAC[NOT_CONS_NIL; HD]]]
```

Note The A1 - A4 in the above tactic are the axioms A1 - A4. Also in axiom A1, *INSERT* adds an element to a set. Thus $[x]$ *INSERT* $\{\}$ is the singleton set containing $[x]$.

THIS PAGE IS LEFT BLANK INTENTIONALLY

Appendix D

The rules added to B for the condense proof

THEORY con
$con\ [x] = [x]$
$x = head(l) \Rightarrow con(x{:}l) = con(l)$
$x \neq head(l) \Rightarrow con(x{:}l) = x{:}con(l)$

THEORY pc
$pc\ [x] = \{[x]\}$
$x = head(l) \Rightarrow pc(x{:}l) = x::pc(l) \cup pc(l)$
$x \neq head(l) \Rightarrow pc(x{:}l) = x::pc(l)$

THEORY distcons
$(l \in x::S) = \exists n.(l = x{:}n \wedge n \in S);$

$(P \Rightarrow Q)$
$\Rightarrow$
$((\exists n.P) \Rightarrow Q)$

THEORY hd
$head(x{:}l) = x$

THEORY cases
$inhyp(m \in pc(x{:}l)) \wedge$
$(x = head(l) \Rightarrow g) \wedge$
$(x \neq head(l) \Rightarrow g)$
$\Rightarrow$
g

$inhyp(l \in S \cup T) \wedge$
$(l \in S \Rightarrow g) \wedge$
$(l \in T \Rightarrow g)$
$\Rightarrow$
g

THEORY listinduct
$[l := [x]]P \wedge$
$\forall l.(P \Rightarrow \forall x.[l := x{:}l]P)$
$\Rightarrow$
$\forall l.P$

THEORY undisch
inhyp(l ∈ x::S) ∧
(l ∈ x::S ⇒ g)
⇒
g

THEORY singleset
$m \in \{l\} \Rightarrow m = l$

THEORY equality1
$S = T \wedge m \in S \Rightarrow m \in T$

THEORY equality2
$x = y \Rightarrow z{:}x = z{:}y$

THEORY prv
(vrb(l,m) ⇒ ∀l ∀m g) ∧
results+g
⇒
prove1(vrb(l,m) ⇒ ∀l ∀m g)

THEORY prove
prove1(vrb(l,m) ⇒ ∀l ∀m (m ∈ pc(l) ⇒ head(m) = head(l)))
prove1(vrb(l,m) ⇒ ∀l ∀m (m ∈ pc(l) ⇒ con(m) = con(l)))
TAC
prv;listinduct;(con;pc;cases;hd;undisch;distcons;equality2)~
FTAC
(results;con;pc;singleset;equality1)~

Note

The theory *prove*, contains the lemma and theorem to be proved. The expression below the keyword *TAC* is the tactic, and the expression below the keyword *FTAC* is the forward tactic. Such a theory does not always have to be called *prove*, it is simply a name chosen by the user. Also, the second rule in the theory *distcons* should have a guard, which ensures that the existentially quantified variable n does not appear free anywhere else in the goal (apart from in P). That is, it should not appear in the hypotheses and it should not appear in the conclusion. It was not clear how to write this guard in B (see section 6.3), and so it has been left out. However, when this rule was used, a manual check was carried out, by the user, to ensure that n was not present anywhere else in the goal.

Appendix E

A description of the m-NEVER commands used in this report

equality substitute. If the expression $E = F$ appears in the expression being simplified, then the command *equality substitute E* will replace appropriate occurrences of E with F. If no expression is supplied, then m-NEVER uses a heuristic to automatically substitute expressions. For example

equality substitute x on $x = 5 \wedge y = 7*x$ gives $x = 5 \wedge y = 7*5$

simplify. This performs the simplification of first order predicate expressions, with propositional tautologies always detected. It also reasons about integers, carries out equality substitutions automatically, and instantiates variables in quantified expressions. Below are some examples

simplify on $\neg\exists x \neg P(x)$ gives $\forall x P(x)$

simplify on $(P \vee Q) \wedge (\neg P \vee R) \Rightarrow (Q \vee R)$ gives *TRUE*

simplify on $x = 5 \wedge y = 7*x$ gives $x = 5 \wedge y = 35$

simplify on $\forall x(x > 5)$ gives *FALSE* (where x is of type integer)

reduce. This is the most powerful of all the commands in m-NEVER. It performs simplification, rewriting and invocation. Invocation includes the replacement of a function for its body. For example if f and g are functions on integers, with f defined using the rewrite rule $f(x) \text{—>} x+1$, and g defined by giving it the body *begin x+2 end*, then

reduce on $\forall x(2*f(x) + 3*g(x) = 5*x + 8)$ gives *TRUE*

prenex. Tries to convert the expression, as close as possible, to the form Q.body, where Q is a list of quantifiers and body is a quantifier free expression.The body is not always quantifier free, because for example, *prenex* does not seem to rename bound variables. Below are some examples:

prenex on $\forall x(x > 5) \vee \forall y(y > 1) \vee \forall z(z < 3)$ gives $\forall x,y,z\,(x > 5 \vee y > 1 \vee z < 3)$

prenex on $\exists x(x = 6) \vee \forall y(y < 1)$ gives $\exists x \forall y(x = 6 \vee y < 1)$

prenex on $\exists x(x = 6) \vee \forall x(x < 1)$ produces no change

open. If the expression is of the form $\forall v_1 ,...,v_n P$ then the command *open* produces the expression P. This command can sometimes be useful after the *prenex* command. Below are some examples:

open on $\forall x \exists y \forall z (x > 0 \vee y < 0 \Rightarrow z = 6)$ gives $\exists y \forall z (x > 0 \vee y < 0 \Rightarrow z = 6)$

open on $\forall x,y \forall z (x > 0 \vee y < 0 \Rightarrow z = 6)$ gives $\forall z (x > 0 \vee y < 0 \Rightarrow z = 6)$

split. The command *split P* where P is a predicate, forms a case split; one case assuming P is true and another assuming P is false. If E is the expression being simplified then the new expression *if P then E else E end if* is formed. This is a good example of how sometimes m-NEVER presents the user with an expression in the conditional form *if_then_else_endif*. An example of the use of *split* is

split $x=7$ on $x > 0 \wedge x < 9$ gives
if $x = 7$ *then* $x > 0 \wedge x < 9$ *else* $x > 0 \wedge x < 9$ *endif*

use. The command *use axiom_name* on the expression E, where *axiom_name* is the name of an axiom, produces the new expression $A \Rightarrow E$, where A is the axiom with its free variables universally quantified. The command can also be used with particular instantiations. Also it can be used with a function to obtain its underlying axiom. Below are some examples, using the axiom named *double1* which is the rewrite rule *double(x)* —> *2*x*.

use double1 on $y < 5$ gives $\forall x(double(x) = 2*x) \Rightarrow y < 5$

use double1 $x = 9$ on $y < 5$ gives $(double(9) = 2*9) \Rightarrow y < 5$

back. The command *back n* takes the user back n steps.

disjunctive. Converts the expression into disjunctive normal form, that is, an expression in the form $(A \wedge B \wedge C \wedge ...) \vee (D \wedge E \wedge F \wedge ...) \vee ...$, where each of A, B, C, D, E, F, ... is either an atomic expression, or the negation of an atomic expression. For example

disjunctive on $(A \vee B) \Rightarrow (C \wedge D)$ gives $(\neg A \wedge \neg B) \vee (C \wedge D)$

induct. Consider the expression $x \geq 0 \Rightarrow factorial(x) \geq 1$, where the function *factorial* has been defined recursively, with precondition $x \geq 0$, and body *if x = 0 then 1 else x*factorial(x-1) endif*. The command *induct* on the original expression will produce the new expression

$[\neg(x \geq 0 \wedge x \neq 0) \Rightarrow (x \geq 0 \Rightarrow factorial(x) \geq 1)] \wedge$
$[(x \geq 0 \wedge x \neq 0 \wedge (x\text{-}1 \geq 0 \Rightarrow factorial(x\text{-}1) \geq 1)) \Rightarrow (x \geq 0 \Rightarrow factorial(x) \geq 1)]$

The command has noticed that the original expression $x \geq 0 \Rightarrow factorial(x) \geq 1$ contains a recursive function, namely *factorial*, and has produced a new expression which captures both the base case and the step case of a proof by induction. The first conjunct is the base case, and the second conjunct is the step case. The predicate $x \geq 0 \wedge x \neq 0$ in the step case, is the precondition together with the negation of the test condition in the body of the function, and thus forms the condition that the recursive part of the body is entered. The rest of the step case is then formed by assuming the original expression holds for $x\text{-}1$, and saying that it must hold with x.

The base case is formed by saying that the original expression must hold when the recursive part of the function body is not entered. The command is a good example of how the user always has just one expression, ie induct does not produce two expressions. *induct* applies a heuristic to the expression to look for recursive functions. However the user can tell m-NEVER to induct on a particular function. There is also a command *prove by induction* which will apply a heuristic to look for recursive functions, set up the new expression which captures a proof by induction, and then perform some simplification. Out of interest, *prove by induction* on the original expression $x \geq 0 \Rightarrow factorial(x) \geq 1$ above returns *TRUE*.

THIS PAGE IS LEFT BLANK INTENTIONALLY

REPORT DOCUMENTATION PAGE

DRIC Reference Number (If known) .....

Overall security classification of sheet .....Unclassified.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
MEMO 4430	OCTOBER	1990

Originators Name and Location

RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title

WHICH THEOREM PROVER?
(A SURVEY OF FOUR THEOREM PROVERS)

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
SMITH, A	32

Abstract

When using Formal Methods to produce verified software, mathematical theorems arise which need to be proved. This memorandum contains the experiences gained in using four theorem provers to prove such theorems. From this experience, a number of recommendations are made on what constitutes a good theorem prover.

Abstract Classification (U,R,C or S)

U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED

THIS PAGE IS LEFT BLANK INTENTIONALLY