Program Verification using HOL-UNITY

Flemming Andersen and Kim Dam Petersen and Jimmi S. Pettersson

Tele Danmark Research, Lyngsø Allé 2, DK-2970 Hørsholm

Abstract. HOL-UNITY is an implementation of Chandy and Misra's UNITY theory in the HOL88 and HOL90 theorem provers. This paper shows how to verify safety and progress properties of concurrent programs using HOL-UNITY. As an example it is proved that a lift-control program satisfies a given progress property. The proof is compositional and partly automated. The progress property is decomposed into basic safety and progress properties, which are proved automatically by a developed tactic based on a combination of Gentzen-like proof methods and Pressburger decision procedures. The proof of the decomposition which includes induction is done mechanically using the inference rules of the UNITY logic implemented as theorems in HOL. The paper also contains some empirical results of running the developed tactic in HOL88 and HOL90, respectively. It turns out that HOL90 in average is about 9 times faster than HOL88. Finally, we discuss various ways of improving the tactic.

1 Introduction

This paper presents an approach to verify safety and progress properties of concurrent programs using HOL-UNITY [2, 3], an implementation of Chandy and Misra's UNITY theory [6] in the HOL theorem provers [12, 20]. By combining a representation of UNITY programs in HOL and the HOL-UNITY theory, it is possible in a structured way to prove that UNITY programs satisfy required properties.

Various aspects of this approach is demonstrated by proving that a lift control program satisfies a given (**leadsto**) progress property. The example distinguishes itself by containing both boolean, arithmetic and quantified program constructs. The progress property formalises that every request for the lift is eventually serviced. To prove this kind of property, it is necessary to prove a number of basic properties satisfied by the transitions of the program [6, 16]. The property is then proved from the basic properties using the inference rules of UNITY logic supported as theorems in HOL-UNITY. Proving that a program satisfies the basic properties is tedious. This is due to the high degree of details in the proofs and the need for interaction with the verifier caused by the lack of automated support in HOL-UNITY. In order to relieve the verifier, automated verification methods needs to be applied.

Based on a combination of Gentzen and Pressburger methods [5, 9, 18, 21] a straight forward implementation of a proof tactic is used to automatically prove the basic properties of the lift control program. Some of the design decisions when making the tactic and consequences thereof are discussed and proposals are given that may be used for improvement.

The paper is organized as follows. In Section 2, UNITY and its implementation in HOL are briefly described. In Section 3, the example is described and the UNITY program for controlling the lift is presented together with its representation in HOL. Section 4 presents the decomposition of the required progress property into basic properties which are provable by the implemented tactic. Section 5 presents an overview of the developed tactic. Finally, section 6 contains some empirical results of verifying the lift example in both HOL88 and HOL90. It turns out that HOL90 in average is 9 times faster than HOL88 for this example. Section 7 concludes the paper discussing what remains to be done in order to achieve a tool applicable to a larger class of examples.

2 HOL-UNITY

UNITY [6] is a theory for specifying and verifying programs. The theory consists of a simple language for specifying programs, a logic to express safety and progress properties which a program must satisfy, and a proof system for proving that programs satisfy such properties. This section presents the formalization of UNITY in the higher order logic [7] supported by HOL [12].

Programs

UNITY programs basically consists of a set of variables, a set of *initialization* equations on the program variables, and a set of conditionally enabled multiple assignments, *actions*. Programs are modeled by (unlabeled) transition systems, which consists of a state space Σ, defined by the variables, the initial condition given by the state predicate init, and a set of actions $\mathcal{A}$.

A state $\sigma \in \Sigma$ is a function from variable names to their respective values. Variable names are represented by the values of an enumeration type corresponding to their unique naming. Values are represented by a type which is defined as the disjoint union of the types given by the variables.

For a program with a boolean variable b and an array variable a of booleans, the type of variable names are defined using Tom Melham's [15] type package: $var = b \mid a$ and the type of values by: $val = Bool\,\mathbf{bool} \mid Array\,(\mathbf{num} \rightarrow \mathbf{bool})$. For each value a destructor is introduced, e.g., $destBool\,(Bool\ x) = x$ such that extracting the value of variables in a state σ can be done by $destBool\,(\sigma\ b)$. Hence, a state dependent variable v_* of type T is defined as: $v_*\ \sigma = destT\,(\sigma\ v)$, where the suffix priming $*$ represents the state lifting of the variable name v. Evidently all program constructs become state lifted. For readability HOL-UNITY defines a collection of state lifting operators. A constant c may be state lifted by prefixing it with the predefined HOL combinator $\mathbf{K}$, which satisfies: $\mathbf{K}\ c\ \sigma = c$. The logic connectors, e.g. $\wedge, \vee, \Rightarrow$ are similarly lifted to state abstracted connectors: $\wedge_*, \vee_*, \Rightarrow_*$ e.g.: $(p \wedge_* \ q)\ \sigma\ = p\ \sigma \wedge q\ \sigma$ and, for array variables the indexing operator **at** is defined as: $(a \textbf{ at } e)\ \sigma = (a\ \sigma)\ (e\ \sigma)$.

Assignments

A (multiple) assignment in UNITY consists of updating a state, by changing the values of one or more of its variables or array elements. If all variables are simple (i.e. non-arrays) the assignment may be performed by first building a list of variable names and computed values and next building the new state by stepping through the list. It is defined by the combinator **asg**, using the notation: $(b\ \rightarrow\ c\ \mid\ e)$ denoting the conditional expression: **if** b **then** c **else** e:

$\textbf{asg}\ ((v', e') :: l)\ \sigma\ v = (v = v')\ \rightarrow\ e'\ \sigma\ \mid\ \textbf{asg}\ l\ \sigma\ v$

If several array elements are to be assigned to the same array variable in one assignment, they must be collected such that the updated array reflects all changes assigned to the variable. The operator **upd** supports multiple update of an array at multiple element positions:

$(a \textbf{ upd } ((i', e') :: l))\ \sigma\ i = (i\ \sigma = i'\ \sigma)\ \rightarrow\ e'\ \sigma\ \mid\ (a \textbf{ upd } l)\ \sigma\ i$

Notice, **asg** and **upd** returns the old state for empty lists. As an example the UNITY assignment: $b,\ a[0],\ a[1]\ :=\ true,\ a[1],\ a[0]$ is represented in HOL-UNITY by:

$\textbf{asg}\,[(b,\ Bool\ \circ (K\,T));$
$\quad (a,\ Array \circ (a_* \textbf{ upd } [((K\,0),\ a_* \textbf{ at } (K\,1));\ ((K\,1),\ a_* \textbf{ at } (K\,0))]))]$

Actions

A UNITY action is a conditionally enabled multiple assignment. The representation of a multiple assignment without enabling conditions has been shown above. The conditional enabling of a multiple assignment is defined in terms of the combinator **when**:

$(a \textbf{ when } g)\ \sigma = g\ \sigma\ \rightarrow\ a\ \sigma\ \mid\ \sigma$

The presented definitions are sufficient to model all language constructs possible in the UNITY programming language of which the variable types are naturally extended to the types in HOL.

Properties

The UNITY logic contains two safety properties: **unless** and **invariant** and two progress properties: **ensures** and **leadsto**. The logic properties in UNITY can be formally defined in terms of modal relations between sets of states expressed by state dependent predicates, where a state dependent predicate p is a function from states to booleans. The three basic properties **unless**, **invariant** and **ensures** are defined for a given program $F = (\Sigma,\ \mathsf{init},\ \mathcal{A})$ as follows.

$$\begin{array}{lll}
p \textbf{ unless } q & \equiv \forall\alpha\colon\ \{p \wedge \neg q\}\ \alpha\ \{p \vee q\} \\
 & \equiv \forall\alpha\colon\ \forall\sigma\colon\ p\ \sigma\ \wedge\ \neg q\ \sigma \Rightarrow p\ (\alpha\ \sigma)\ \vee\ q\ (\alpha\ \sigma) \\
p \textbf{ invariant} & \equiv (\mathsf{init} \Rightarrow p)\ \wedge\ \forall\alpha\colon\ \{p\}\ \alpha\ \{p\} \\
 & \equiv \forall\sigma\colon\ (\mathsf{init}\ \sigma \Rightarrow p\ \sigma)\ \wedge\ (\forall\alpha\colon\ \forall\sigma\colon\ p\ \sigma \Rightarrow p\ (\alpha\ \sigma)) \\
p \textbf{ ensures } q & \equiv p \textbf{ unless } q\ \wedge\ \exists\alpha\colon\ \{p \wedge \neg q\}\ \alpha\ \{q\} \\
 & \equiv p \textbf{ unless } q\ \wedge\ (\exists\alpha\colon\ \forall\sigma\colon\ p\ \sigma\ \wedge\ \neg q\ \sigma \Rightarrow q\ (\alpha\ \sigma))
\end{array}$$

where α ranges over the set of actions in the program ($\alpha \in \mathcal{A}$), init is the predicate which characterizes the initial states, and σ ranges over the complete state space of the program, i.e., $\sigma \in \Sigma$. The progress property **leadsto** is defined inductively as the transitive, disjunctive closure of **ensures** properties, i.e., it is defined as the least fixed point solution to the following three implications [3, 4]:

$$\begin{array}{ll}
p \textbf{ ensures } q & \Rightarrow p \textbf{ leadsto } q \\
(\exists r\colon\ p \textbf{ leadsto } r \wedge r \textbf{ leadsto } q) & \Rightarrow p \textbf{ leadsto } q \\
(\exists P\colon\ (p = \bigvee P) \wedge (\forall p' \in P\colon\ p' \textbf{ leadsto } q)) & \Rightarrow p \textbf{ leadsto } q
\end{array}$$

where $\bigvee P$ is the disjunction of all the elements in the possibly infinite set of predicates P.

The UNITY logic also includes a number of inference rules proved as theorems in HOL-UNITY. These rules allow us to derive new properties from other properties. E.g., to prove that a program satisfies a specific **leadsto** property we must find some basic properties (unless, invariant and ensures properties) which can be proved from the individual actions of the program. From these basic properties the **leadsto** property can then be proved using the inference rules mentioned above. Hence, we have two kinds of proofs in UNITY: (1) proofs of the basic properties which amounts to proving assertions over individual actions, and (2) proofs of properties derived from other properties. Doing proof by hand, proofs of the first kind are usually left out, the basic properties are just assumed to be valid. However, using HOL requires proving both.

3 The lift-control program

Here the lift-control program is presented (Figure 1). The system consists of a lift that moves between a number of floors to serve requests on these. A request on a floor is served by eventually moving to and stopping the lift at the floor and opening the doors. The bottom and top floors are indicated by two parameters *min* and *max*, satisfying $min \leq max$.

The necessary elements of a lift to reflect its behavior are: floor position, door- and movement conditions, and request indicators. The actions of the lift can be represented by assigning values to variables, e.g., opening the door by setting a variable *open* to true. The state space of the lift can be represented by 6 variables: *floor* (the current position of the lift), *open* (whether the door is open at *floor*), *stop* (whether the lift is stopped at *floor*), *req* (for each floor whether the lift is requested at that floor), *up* (the current direction of movement for the lift), and *move* (whether moving the lift takes precedence over opening of the doors). The first four variables defines the *observable* parts of the lift. The two remaining variables have been introduced to enable a *control* of the lift that ensures fair progress of the lift as discussed below.

The progress requirement of the lift is that any request is eventually satisfied. The strategy for achieving this is to let the lift continue in one direction as long as there are requests to be served in that direction. During this movement, the lift will serve the floors with requests. When there are no more requests in the current direction, the lift starts to move in the other direction, provided there are any in either direction outstanding requests in that direction. When no requests are outstanding the lift is stopped until new requests are made. To avoid a continued request on a floor from blocking the lift at that floor, the moving of the lift to another floor with a request takes precendence over re-opening the door. The variable *move* is used to indicate this.

```
  program Lift(min,max)
    declare
      floor                 : min..max
      up,move,stop,open     : bool
      req                   : array[min..max] of bool
    initially
      floor                 = min
[t]   up,move,stop,open     = false,true,true,false
      ⟨∀i ∈ min..max: req[i] = false⟩
    always
      above                 = $\exists i$: $floor < i \le max \wedge req[i]$
      below                 = $\exists i$: $min \le i < floor \wedge req[i]$
      queueing              = above $\vee$ below
      goingup               = above $\wedge$ ($up \vee \neg$below)
      goingdown             = below $\wedge$ ($\neg up \vee \neg$above)
      ready                 = $stop \wedge \neg open \wedge move$
  assign
    stop,move             := true,false          when $\neg stop \wedge req[floor]$                (1)
  [] open,req[floor],move := true,false,true     when  $stop \wedge \neg open \wedge req[floor] \wedge$
                                                      $\neg(move \wedge$ queueing)                  (2)
  [] open                 := false               when  $open$                                      (3)
  [] stop,floor,up        := false,floor + 1,true when ready $\wedge$ goingup                       (4)
  [] stop,floor,up        := false,floor − 1,false when ready $\wedge$ goingdown                    (5)
  [] floor                := floor + 1           when $\neg stop \wedge\ \ up \wedge \neg req[floor]$   (6)
  [] floor                := floor − 1           when $\neg stop \wedge \neg up \wedge \neg req[floor]$ (7)
end
```

Fig. 1. UNITY program for lift control

Initialization

Basically the *observable* variables should reflect "physically" valid values, and the *control* variables should be set properly. Hence, we assume the lift is initially in a state where it is stopped at the bottom floor without any requests. This is reflected by the **initially** clause of the program in Figure 1.

Actions

The lift is controlled by a collection of *actions* prescribing the updating of its state. Each action is conditionally *enabled* by a predicate, which indicates when the action can change the state. The lift consists of 7 actions (Figure 1): (1) stops the lift when the present floor has a request, (2) opens the door, when the lift is stopped at a floor with request, and opening request has precedence over moving, (3) closes the door, (4) and (5) starts the lift in the presence of a request in up- or downwards direction, when the doors are closed and moving takes precedence over opening them, and (6) and (7) continues moving the lift when it is already in motion and there is no request at the present floor.

Representation of the program in HOL-UNITY

The types representing the lift variable names, and their values are:

$var = floor \mid up \mid move \mid stop \mid open \mid req$
$val = Bool\ \mathbf{bool} \mid Num\ \mathbf{num} \mid Bits\ (\mathbf{num} \rightarrow \mathbf{bool})$
from which the state dependent variables $floor_*$, up_*, $move_*$, $stop_*$, $open_*$ and req_* are defined as described above, e.g., the $floor_*$ variable:

$floor_*\ \sigma = destNum\,(\sigma\ floor)$
The initial condition is defined by the predicate:

$$\begin{aligned} init = {} & (floor_* =_* \mathbf{K}\,min) \wedge_* (up_* =_* \mathbf{K}\,\mathbf{F}) \wedge_* (move_* =_* \mathbf{K}\,\mathbf{T}) \wedge_* \\ & (stop_* =_* \mathbf{K}\,\mathbf{T}) \wedge_* (open_* =_* \mathbf{K}\,\mathbf{F}) \wedge_* \\ & (\forall_* i\colon\ req_*\ \mathbf{at}\,(\mathbf{K}\,i) =_* \mathbf{K}\,\mathbf{F}) \end{aligned}$$

Each individual program action is represented in HOL-UNITY as a conditionally enabled multiple assignment. Action (1), for example, is represented by:

$$\begin{aligned} & \mathbf{asg}\,[(stop,\ Bool \circ (\mathbf{K}\,\mathbf{T})); \\ & \quad (move,\ Bool \circ (\mathbf{K}\,\mathbf{F}))]\ \mathbf{when}\ (\neg_* stop_* \wedge_* req_*\ \mathbf{at}\ floor_*) \end{aligned}$$

In this way, the complete program is defined as a list of **asg** actions.

4 Mechanized proof of progress property

The progress property which we will prove is the requirement that any request for service will eventually be served. In UNITY logic this property can be formalized as:

$\forall n: \ min \le n \le max \Rightarrow req[n]$ **leadsto** $open \wedge floor = n$

It turns out that the lift-control program *Lift* does not have this property in HOL-UNITY. The problem is that in HOL-UNITY, the **leadsto** property must be satisfied for any state in the *total* state space spanned by the program variables. The progress property, however, is only required to be satisfied for *reachable* states of the program. This means, that the lift program may only guarantee the HOL-UNITY progress for a subset of the total state space. This subset which we will call the *valid* states includes the reachable states of the program.

The valid states may be represented by a predicate. Using this predicate, a strengthened **leadsto** property can be formalized and proved in HOL-UNITY:

$\forall n: \ min \le n \le max \Rightarrow req[n] \wedge valid$ **leadsto** $open \wedge floor = n$

In the following a characterization of the *valid* predicate adequate for proving the given **leadsto** property of the *Lift* program is described.

Valid state space

The *valid* states of a program represented by a predicate may be described as a conjunction of predicates, each characterizing a requirement to the acceptable states of the lift program. This also implies that a *valid* predicate must be a program invariant. In UNITY, this requirement is expressed as:

valid **invariant**

In general, the conjuncts in the *valid* predicate may be classified as the different properties: (a) Directly given safety requirements such as lower and upper bound of variables (b) Implicit given safety dependency requirements describing value dependencies between variables and (c) Builtin fairness properties which guarantee progress.

The valid *Lift* states is the conjunction of the following predicates:

(1) $min \le floor \le max$, the lift must be within bottom and top floors.

(2) $open \Rightarrow stop$, the lift must be stopped when the doors are open.

(3) $open \Rightarrow move$, a floor has been served when the doors are open.

(4) $stop \wedge \neg move \Rightarrow req[floor]$, when the lift is stopped and *floor* has not been served, there must be a request on *floor*.

(5) $\neg stop \wedge up \Rightarrow \exists f: \ floor \le f \le max \wedge req[f]$, when the lift is moving upwards, there must be a request for service in this direction.

(6) $\neg stop \wedge \neg up \Rightarrow \exists f: \ min \le f \le floor \wedge req[f]$ when the lift is moving downwards, there must be a request for service in that direction.

In this conjunct (1) belongs to class (a), conjuncts (2), (4), (5) and (6) belong to (b) and (3) belongs to class (c). We are now ready to present a method for proving the strengthened leadsto property.

Progress Decomposition

The first step in proving the leadsto property is to decompose it into simpler UNITY properties from which the required leadsto property can be deduced using the transitivity, disjunctivity and derived inference rules for leadsto, e.g. the following variant of the disjunction rule called case split:

$$\frac{p \wedge b \textbf{ leadsto } q\ ,\ p \wedge \neg b \textbf{ leadsto } q}{p \textbf{ leadsto } q}$$

For displaying the decomposition we use a generalized version of Owicki-Lamport proof-lattices [16]. A UNITY proof lattice for the wanted progress property is shown in Figure 2.

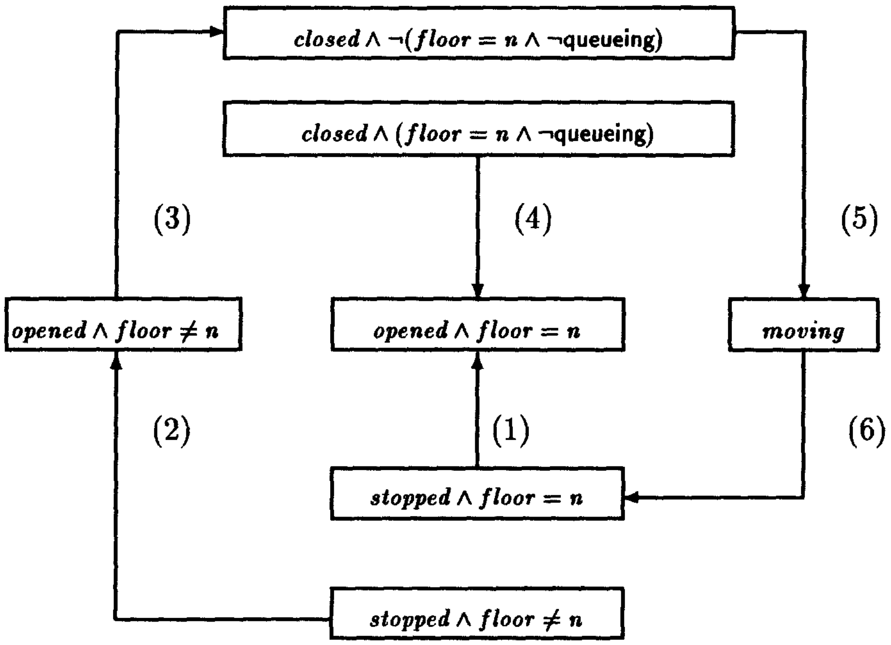


Fig. 2. Proof lattice for the progress property

The arrows in the figure represent leadsto properties to be proved, and the nodes are state predicates just as in the Owicki-Lamport method. However, the directed acyclic graph making up a UNITY proof lattice does not necessarily have a single entry node. In fact, for the proof lattice in Figure 2 every node is a possible entry node. The proof lattice has only one exit node though, just like in Owicki and Lamports proof lattices. Owicki and Lamport describe how invariants propagate through leads-to properties in proof lattices for temporal logic [16]. This result also applies to UNITY logic.

The property $valid \wedge req[n]$, is assumed to be conjuncted to all predicates in the proof lattice. Hence, an arrow: $p \wedge valid \wedge req[n]$ **leadsto** $q \wedge req[n]$ is written as: p **leadsto** q.

We have partitioned the valid states into four classes in the proof lattice: *moving* (the lift is moving), *stopped* (the lift is stopped, but the door has not been opened yet), *opened* (the door has been opened after the lift is stopped), and *closed* (the door has been closed again, but the lift has not started moving yet). The predicates describing these classes are given below.

$$\begin{aligned}
moving &= \neg stop \wedge \neg open \\
stopped &= \ \ stop \wedge \neg open \wedge \neg move \\
opened &= \ \ stop \wedge \ \ open \wedge \ \ move \\
closed &= \ \ stop \wedge \neg open \wedge \ \ move
\end{aligned}$$

The proof lattice actually describes the proof of the **leadsto** property:

$$(moving \vee stopped \vee opened \vee closed) \wedge valid \wedge req[n] \ \textbf{leadsto} \ opened \wedge floor = n$$

from which the wanted property can be easily derived. The arrows (1) to (4) in Figure 2 all follow from corresponding **ensures** properties. The arrow (5) is proved by a state split into the cases goingup and its negation; each of these cases follows from corresponding ensures properties. The arrow (6) is proved by induction as discussed below.

Metric Induction

The metric induction principle of UNITY is based on the traditional decreasing metric technique, which is formally expressed by the inference rule:

$$\frac{p \wedge (Metric = N) \ \textbf{leadsto} \ (p \wedge (Metric < N)) \vee q}{p \ \textbf{leadsto} \ q}$$

The leadsto property represented by the arrow (6) in the proof lattice of Figure 2, viz.,

$$moving \ \textbf{leadsto} \ stopped \wedge floor = n$$

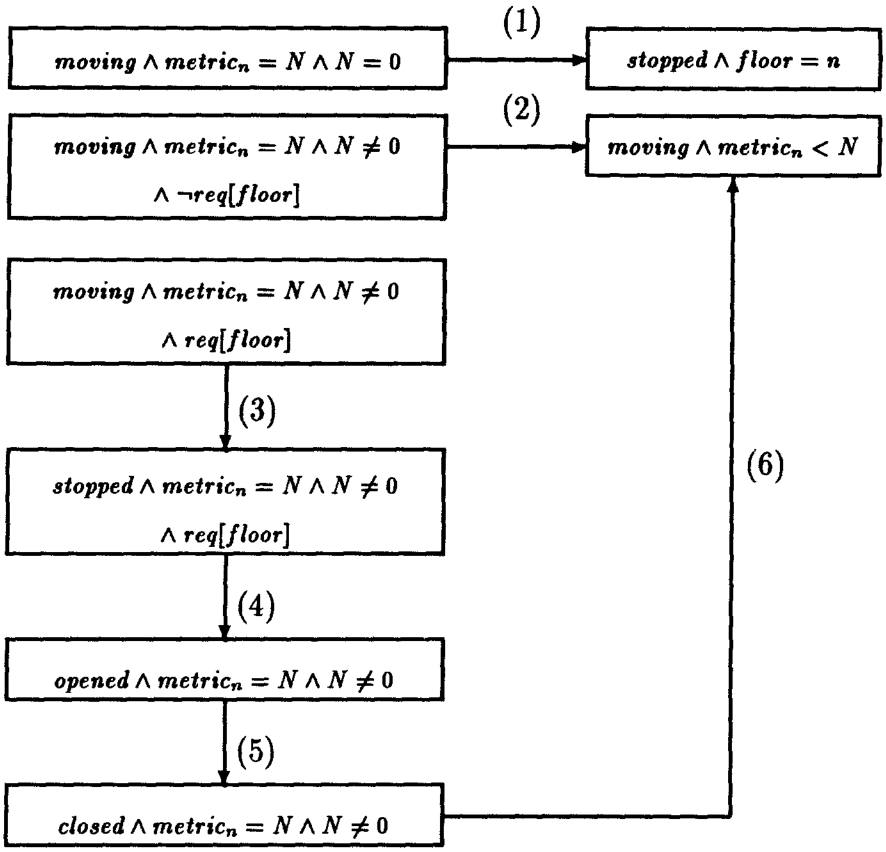
can using this induction principle be deduced from the following induction property:

$$moving \wedge (metric_n = N) \ \textbf{leadsto} \ (moving \wedge metric_n < N) \vee (stopped \wedge floor = n)$$

The predicate $valid \wedge req[n]$ is just as in Figure 2, implicitly conjuncted to the predicates of the property. The metric function $metric_n$ denotes the longest distance to move from the *floor* where the lift is currently located to the n'th floor counted in numbers of floors. $metric_n$ is defined as the worst case number of floors the lift has to move before it arrives at the wanted floor:

$$\begin{aligned}
metric_n = \ & (up \wedge floor < n) \ \rightarrow (n - floor) \ \mid \ (\neg up \wedge n < floor) \rightarrow (floor - n) \\
\mid \ & (up \wedge n < floor) \ \rightarrow (max - floor) + (max - n) \\
\mid \ & (\neg up \wedge floor < n) \rightarrow (floor - min) + (n - min) \ \mid \ 0
\end{aligned}$$

A proof-lattice for the induction property is given in Figure 3. For this proof



$moving \wedge metric_n = N \wedge N = 0$
(1)
$stopped \wedge floor = n$
$moving \wedge metric_n = N \wedge N \neq 0 \wedge \neg req[floor]$
(2)
$moving \wedge metric_n < N$
$moving \wedge metric_n = N \wedge N \neq 0 \wedge req[floor]$
(3)
$stopped \wedge metric_n = N \wedge N \neq 0 \wedge req[floor]$
(4)
$opened \wedge metric_n = N \wedge N \neq 0$
(5)
$closed \wedge metric_n = N \wedge N \neq 0$
(6)

Fig. 3. Proof lattice for induction property

lattice the three nodes in the top left corner of the figure are the possible entry nodes as the disjunction of the three predicates is equivalent to the predicate

$moving \wedge metric_n = N$

and the two nodes in the upper right corner are the exit nodes as the disjunction of the two predicates is equal to the consequence of the induction property.

The arrows (1), (3), (4) and (5) in Figure 3 follow directly from corresponding ensures properties. The arrow (2) is proved using state split considering the cases: *up* and its negation; the two cases follow from corresponding ensures properties. The arrow (6) is proved using state split considering the cases: goingup and its negation; again the cases follow from corresponding ensures properties.

It can be checked by HOL-UNITY that these proof lattices actually form a proof of the wanted leadsto property, i.e., that the leadsto property can be derived from the ensures properties present as arrows in the proof lattices using the derived inference rules in HOL-UNITY, including the rule induction. Proving that the lift-control program actually satisfies the ensures properties can be automated to a large degree, as described in the next section.

5 A Program Property Tactic

HOL-UNITY is a higher order logic theory, since UNITY is defined as a theory within a typed higher order logic. However, by restricting the expressions of HOL-UNITY programs to consist of state lifted boolean, number or array values, a HOL-UNITY property that must be satisfied by programs is in fact restricted to first order logic extended with arithmetic relations.

As a consequence, proving the program properties **unless**, **invariant** and **ensures** in HOL-UNITY can be automated by first eliminating the UNITY operators, then simplifying the state lifted operators, and finally applying standard Gentzen-like techniques attempting to prove the logic parts of the proof goal, and if this fails applying the Pressburger decision procedure on the arithmetic relations.

Based on this idea a tactic is developed for automating the proof of such properties. An overview of some simple strategies applied by the proof tactic to speed up the proving is presented.

Reduction of HOL-UNITY properties

Every **unless**, **invariant** and **ensures** property that must hold for a program, i.e. is applied to the program which is represented as a list of (**asg**) state transition actions can be rewritten into a conjunction of properties each applied to only one action. Utilizing this knowledge we can reduce the size of the goals to be proved and obtain a proof algorithm which is linear in the number of actions in the program. In addition to eliminating the operators by inserting their definitions, we may further reduce the terms when applying a conditionally enabled action a **when** g to the properties:

- The property p **unless** q is reduced to:
  $(p \wedge_* \neg_* q \wedge_* g) \Rightarrow_* ((p \vee_* q) \circ a)$
  where it is utilized that a is a function from states to states.
- The property **invariant** p is reduced to:
  $(\mathsf{init} \Rightarrow_* p) \wedge ((p \wedge g) \Rightarrow_* (p \circ a))$
- The existential part of p **ensures** q is reduced to:
  $(p \wedge_* \neg_* q \wedge_* g) \Rightarrow_* (q \circ a)$, and $(p \wedge_* \neg_* q \wedge_* \neg_* g) \Rightarrow_* q$

Actions without enabling conditions are reduced similarly with g equal to true.

Example:

- The **unless** property
  $(v = n \wedge 0 \leq v)$ **unless** $(v = n + 1)$ $[v := v + 1$ **when** $0 \leq v]$
  is reduced to the predicate:
  $(v_* =_* (K\, n) \wedge_* (K\, 0) \leq_* v_*) \wedge_* \neg_*(v_* =_* (K\, n) +_* (K\, 1)) \wedge_* ((K\, 0) \leq_* v_*) \Rightarrow_*$
  $(v_* =_* (K\, n) \wedge_* (K\, 0) \leq_* v_*) \vee_* (v_* =_* (K\, n) +_* (K\, 1))) \circ (\mathbf{asg}[v, v_* +_* (K\, 1)])\, \sigma$

Notice that all operators in the above predicates are state lifted as indicated by the suffix priming $*$.

Reduction of Program Constructs

It is possible to apply reductions to the defined language constructs like for the HOL-UNITY properties. It is assumed that a state is either a variable σ or state derived from a state variable σ by an application $\mathbf{asg}[v_1, e_1; \ldots]\, \sigma$ where v_i are variable names and e_i are expressions in which **asg** does not occur. The following rewrite rules are applied in the tactic:

- For each variable v, in the above $\mathbf{asg}[v_1, e_1; \ldots]\, \sigma$:
  $(v_* (\mathbf{asg}\, ((v, T \circ e) :: l))\, \sigma = e\, \sigma)$
- For each variable u different from v:
  $(v_* (\mathbf{asg}\, ((u, T \circ e) :: l))\, \sigma = v_* (\mathbf{asg}\, l)\, \sigma)$

Example (continued):

- The result predicate presented above is now reduced to:
  $((v_* \sigma = n \wedge 0 \leq v_* \sigma) \wedge \neg(v_* \sigma = n + 1) \wedge (0 \leq v_* \sigma)) \Rightarrow$
  $((v_* \sigma + 1 = n \wedge 0 \leq v_* \sigma + 1) \vee (v_* \sigma + 1 = n + 1))$

Other Reductions

It may be convenient to get rid of subtraction, conditional expressions and arithmetic relations different from $<$. Boulton has automated this process for subtraction and conditional expressions as part of his arithmetic library [5]. Hence, only the inequality operators: $>$, $\leq$ and $\geq$ need to be replaced by terms of $<$, e.g., $a \leq b = a < b + 1$. This is also done by the tactic.

More about the tactic

As a pre-processing, the automation tactic utilizes reduction rules as described above before any attempt is done to use a Gentzen like way of proving a property. If the tactic fails to prove the goal in this way it will apply a strategy, including quantifiers, for proving lower and upper bound cases implied by arithmetic inequalities. Finally if everything else fails the Pressburger decision procedure is applied.

6 Empirical results

The example was originally initiated in HOL88 but completed in HOL90 because it turned out that HOL90 was considerably faster for this kind of proving. Table 6 shows the results in CPU-seconds of executing the proof tactic in HOL88 and HOL90, respectively, on a DECstation 5000/200 running Ultrix V4.2A.

Proof obligation	hol90	hol88	speedup
invariance of valid	97.0	407.4	4.20
(1)	53.7	137.6	2.56
(2)	54.7	159.3	2.91
(3)	76.4	248.4	3.25
(4)	80.9	318.7	3.93
(5)	160.5	472.1	2.94
(1)	93.0	256.7	2.76
(2)	430.2	2751.0	6.39
(3)	102.6	185.0	1.80
(4)	114.4	306.4	2.67
(5)	103.3	237.2	2.29
(6)	1444.7	19433.0	13.45
Total	2811.4	24912.8	8.86

The first line in the table shows the time for proving that the predicate *valid* is invariant. The next 5 lines show the times for proving the ensures properties corresponding to the arrows marked (1) to (5) in the proof lattice of Figure 2. Note that the arrow marked (5) corresponds to two ensures properties. The last six lines in the table shows the times for proving the ensures properties corresponding to the arrows marked (1) to (6) in the proof lattice of Figure 3. Here (2) and (6) corresponds to two ensures properties. These two properties are also induction steps which accounts for their relatively high execution times. The last column of the table shows the number of times that the execution of the tactic in HOL90 is faster than HOL88.

It follows that for invariant and non-induction steps HOL90 is 3-4 times faster than HOL88. However, for the two induction steps HOL90 is 6 and 13 times faster than HOL88. The two induction steps includes the largest amount of arithmetic proofs. In average HOL90 is 9 times faster than HOL88.

7 Discussion

In this paper we have shown how to prove that a lift-control program satisfies a given progress property using HOL-UNITY [3]. We have presented a tactic for automatically proving the basic **unless**, **invariant** and **ensures** properties. From these properties the required **leadsto** property is derived as shown in the proof lattices in Figure 2 and 3 using the inference rules of the UNITY [6] logic built into HOL-UNITY as proved theorems.

Combining a Pressburger proof procedure, recently introduced as a library [5, 13] in the HOL system, with a Gentzen-like way of proving, it has been possible to implement the tactic in HOL enabling automatic proofs of the basic properties of the lift-control example.

The example proof of a single leadsto progress property contains 120 applications of the tactic to prove the necessary ensures and invariance properties satisfied by the transitions of the lift program. Thus, the tactic does a great deal of tedious work for us, enabling us to concentrate on the difficult aspects of the proof, viz. finding the invariant and decomposing the leadsto property. How to find the invariants and decompose the leadsto properties has not been discussed here.

In proving a basic property satisfied by a program we utilized that it is possible to first separately prove the property satisfied for each action of the program and then from these properties derive a proof of the property for the entire program. This proof method is possible, since the basic properties satisfy the theorem:

$$\frac{Pr = [st_1;\ ...;\ st_n],\ R\ p\ q\ [st_1],\ ...,\ R\ p\ q\ [st_n]}{R\ p\ q\ Pr}$$

where R is one of the properties **unless**, **invariant** or **ensures**. Using this method has the advantage that every single proof is of minimum size, but the number of proofs are linear in the number of program actions.

In view of the timing results, this may put limitations on proving properties of larger examples. To speed up the currently implemented tactic, it contains strategies looking for a certain class of contradictions, specialized optimizations for lower and upper bound analysis of arithmetic terms and term normalization. These design decisions speed up the search for a proof in many of the occasions in the example. Another perhaps better solution is adapting the *Faust* prover [17] to combine it with the Pressburger tool.

Our next goal is to finish the ongoing development of a compiler for UNITY. The compiler will support generating the HOL-UNITY representation and the proof obligations. Until it is available, the generation of HOL-UNITY information has to be done manually.

References

1. S. Agerholm. Mechanizing Program Verification in HOL. Master's thesis, Computer Science Department, University of Århus, Denmark, September 1991.
2. F. Andersen. A Definitional Theory of UNITY in HOL. In *Summary of talks at the Third Annual HOL User Meeting*, PB 340, pages 151–162. DAIMI, Århus University, october 1990.
3. F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark, 1992. Also published as TFL RT 1992-3.

4. F. Andersen and K. D. Petersen. Recursive Boolean Functions in HOL. In *1991 International Tutorial and Workshop on the HOL Theorem Proving System and its Applications*, pages 367–377. IEEE Computer Society, August 1991.
5. R. Boulton. The HOL arith Library. Technical report, Computer Laboratory University of Cambridge, July 1992.
6. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
7. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5, 1940.
8. U. Engberg, P. Grønning, and L. Lamport. Mechanical Verification of Concurrent Systems with TLA. In *Fourth International Workshop on Computer Aided Verification*, 1992.
9. J. H. Gallier. *Logic for Computer Science.* Foundations of Automatic Theorem Proving. Harper & Row, Publishers, 1986.
10. S. Garland, J. Guttag, and J. Staunstrup. Verification of VLSI circuits using LP. Technical report, DAIMI PB-258, University of Århus, Denmark, July 1988.
11. D. M. Goldschlag. Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16(9):1004–1023, September 1990.
12. M. J. C. Gordon. *HOL - A Proof Generating System for Higher-Order Logic.* Cambridge University, Computer Laboratory, 1987.
13. J. Harrison. The HOL reduce Library. Technical report, Computer Laboratory University of Cambridge, June 1991.
14. C.S. Jutla, E. Knapp, and J.R. Rao. A Predicate Transformer Approach to Semantics of Parallel Programs. *ACM Symposium on Principles of Distributed Computing*, 1989.
15. T. Melham. Automating Recursive Type Definitions in Higher Order Logic. Technical Report No. 146, Computer Laboratory, University of Cambridge, Sept. 1988.
16. S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3), July 1982.
17. K. Schneider, R. Kumar, and T. Kropf. New Concepts in Faust. In *1992 International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 471–493. *imec* Interuniversity Micro-Electronics Center, September 1992.
18. R. E. Shostak. Deciding Combinations of Theories. *JACM*, 31:1–12, 1984.
19. Beverly A. Sanders. Eliminating the Substitution Axiom from UNITY Logic. *Formal Aspects of Computing*, 3(2):189–205, April-June 1991.
20. K. Slind. HOL90 Users Manual. Technical report, 1992.
21. R. Smullyan. *First Order Logic*, volume 43 of *Ergebnisse der Mathematik und ihrer Grenzgebiete.* Springer–Verlag, second printing 1971 edition, 1968.