Using a Theorem Prover for Reasoning about Concurrent Algorithms

Joakim von Wright and Thomas Långbacka

Department of Computer Science, Åbo Akademi University
Lemminkäinengatan 14, SF-20520 Turku, Finland

Abstract. An attempt to mechanise reasoning about concurrent algorithms is described. The HOL theorem prover is used to formalise the semantics of the Temporal Logic of Actions (TLA). Using this formalisation, the proof rules of TLA are proved as theorems in the HOL system. The use of HOL in reasoning about algorithms in TLA is illustrated by two examples: a proof of a program property and an implementation proof.

1 Introduction

The Temporal Logic of Actions (TLA) is a logic for reasoning about concurrent algorithms, developed recently by Leslie Lamport [5, 6]. Algorithms are expressed directly as formulas in TLA, rather than in terms of a separate programming language. In this paper we describe an attempt to mechanise TLA reasoning using the theorem prover HOL [4]. Since reasoning about algorithms using TLA involves a lot of tedious proof details, much would be gained if the reasoning could be verified and partly automated using a proof assistant.

The HOL system has mainly been used for hardware verification. However, the higher order features of HOL also makes it useful for formalising logics and programming semantics [1, 2]. The fact that TLA expresses algorithms directly in the logic is an advantage, since this means that there is only one formalism that has to be implemented.

The HOL system is an interactive proof assistant, based on higher order logic (a polymorphic version of the simple theory of types [3]). Other logics can be embedded in HOL, using the HOL logic as a meta-logic. In this paper, which extends the previous work in [8], we formalise TLA in HOL by defining the semantics of TLA in HOL. The proof rules of TLA are then proved as HOL theorems. After this, these rules can be used for reasoning about programs. This means that we can show that a program has a certain property, or that one program implements another one, by proving a corresponding HOL theorem. We restrict ourselves to *Simple TLA*. This means that we do not consider hiding of variables and refinement mappings.

We assume that the reader is familiar with the HOL system and its version of higher order logic, as described in the documentation of the HOL system [4]. When referring to HOL-terms and interaction with the HOL system we use the syntax of HOL. In particular, we note that the scope of binders and quantifiers extends as far to the right as possible. To make formulas more readable, w e often omit type information which can be inferred from the context. Also, we use the ordinary logical symbols (the boolean truth values are denoted F and T).

2 Basic Concepts of TLA in HOL

In this section, we first give an overview of TLA and then describe how TLA is semantically embedded in HOL.

2.1 The Temporal Logic of Actions

One of the main ideas of TLA is that algorithms are expressed directly in the logic, rather than in a separate programming notation. TLA is based on a simple temporal logic, with $\Box$ ("always") as the only primitive temporal operator.

An *action* is a boolean expression that states how initial and final states are related. As an example,

$$(x' = x + 1) \wedge (y' = y) \tag{1}$$

is an action that increments x by 1 and leaves y unchanged (thus primed variables always stand for final states).

Predicates can always be interpreted as actions (as actions, they put no restrictions on the final states). Similarly, both predicates and actions can be interpreted as temporal formulas. A predicate asserts something about the state at time 0 while an action relates the states at time 0 and time 1.

If $\mathcal{A}$ is an action and f is a state function, then $[\mathcal{A}]_f$ is the action

$$[\mathcal{A}]_f \overset{\text{def}}{=} \mathcal{A} \vee (f = f')$$

where f' is the same as f but with all occurrences of program variables primed.

As an example, if $\mathcal{A}$ is the action $(x' = x + 1) \wedge (y' = y)$ then the action $[\mathcal{A}]_{(x,y)}$ permits arbitrary changes of all other variables than x and y. In this way stuttering can be modelled.

Variables and Types. TLA assumes that there is an infinite supply of program variables, though a given algorithm always mentions only a finite number of them. The values that variables can take are not organised in types. Instead, TLA assumes that there exists only one single set of values, and a variable can take any value from this set.

Expressing Algorithms in TLA. An algorithm is described by a formula

$$\Pi = \mathit{Init} \wedge \Box[\mathcal{N}]_f \wedge F \tag{2}$$

where *Init* is a predicate that characterises the permitted initial states, $\mathcal{N}$ is the disjunction of all the actions of the algorithm and F is a formula that describes a fairness condition (thus safety and liveness are treated within a single framework). Note that the predicate *Init* is used as such in the TLA-formula (2). This shows how TLA permits predicates, actions and temporal formulas to be combined.

The safety part of (2) is $\Box[\mathcal{N}]_f$ which states that every step of the program must be permitted by $\mathcal{N}$ or it must leave f unchanged. Typically, f is a tuple containing all variables that the algorithm works on. Then $\Box[\mathcal{N}]_f$ states that every step is an $\mathcal{N}$-step or a stuttering step.

The liveness part F is typically a conjunction of fairness conditions on actions of the algorithm. Fairness requirements are easily expressed in TLA. Weak and strong fairness with respect to an action $\mathcal{A}$ is expressed as

$$\begin{aligned} WF_f(\mathcal{A}) &\overset{\text{def}}{=} (\Box\Diamond\langle\mathcal{A}\rangle_f) \vee (\Box\Diamond\neg Enabled\langle\mathcal{A}\rangle_f) \\ SF_f(\mathcal{A}) &\overset{\text{def}}{=} (\Box\Diamond\langle\mathcal{A}\rangle_f) \vee (\Diamond\Box\neg Enabled\langle\mathcal{A}\rangle_f) \end{aligned}$$

where $\langle\mathcal{A}\rangle_f = \neg[\neg\mathcal{A}]_f$ and $Enabled(\mathcal{A})$ is true of those states in which it is possible to perform $\mathcal{A}$.

As a proof system, TLA has only a small set of basic proof rules. In addition to this, simple temporal reasoning is used, as well as ordinary mathematical reasoning about actions. In TLA, proving that program Π has property $\Box\Phi$ means proving that the TLA formula

$$\Pi \Rightarrow \Box\Phi$$

is valid. Similarly, proving that Π is implemented by another program Π' means proving that the formula $\Pi' \Rightarrow \Pi$ is valid.

An Example Algorithm. As an example we consider a simple algorithm, which increments variables x and y indefinitely. The initial state is characterised by the predicate

$$Init_1 \overset{\text{def}}{=} (x = 0) \wedge (y = 0)$$

The incrementation is defined by the following two actions:

$$\begin{aligned} \mathcal{M}_1 &\overset{\text{def}}{=} x' = x + 1 \wedge y' = y \\ \mathcal{M}_2 &\overset{\text{def}}{=} y' = y + 1 \wedge x' = x \end{aligned}$$

The fairness requirement is weak fairness with respect to both actions, so the algorithm is described by the following formula:

$$\Pi_1 = Init_1 \wedge \Box[\mathcal{M}_1 \vee \mathcal{M}_2]_{(x,y)} \wedge WF_{(x,y)}(\mathcal{M}_1) \wedge WF_{(x,y)}(\mathcal{M}_2) \qquad (3)$$

2.2 States, Predicates and Actions in HOL

In our formalisation, we assume that every variable has a well-defined type. Much can be said about advantages and disadvantages of typing. Our choice is dictated by the typing rules of HOL, and in Sect. 4.2 we will se how the typing supports implementation proofs.

We formalise states as tuples where every component corresponds to one variable of the state (the type of the component indicates what type its values must have). The state is made potentially infinite by adding a final component with polymorphic type. This final component ("the rest of the universe") can be instantiated to any tuple.

As an example, we consider the state space containing the variables x and y which range over natural numbers. The corresponding state space in HOL has type `:num#num#*` (type variables have names beginning with an asterisk).

Predicates and actions are formalised as boolean expressions with lambda-bound program variables. For example, in the above mentioned state space, the action (1) is formalised as

```
λ(x,y,z)(x',y',z'). (x'=x+1) ∧ (y'=y)
```

and the state predicate $x > 0$ is formalised as

```
λ(x,y,z). x>0
```

With this way of treating predicates and actions, substitutions can be formalised neatly as a combination of an application and an abstraction. For example, the predicate $x + y > 0$, with y substituted for x, is formalised as

```
λ(x,y,z). (λ(x,y,z). x+y>0)(y,y,z)
```

which beta-reduces to

```
λ(x,y,z).y+y>0
```

as it should. Note that the variables are anonymous, in the sense that the action (1) is equivalently expressed by

```
λ(a,z,x)(c,t',z'). (c=a+1) ∧ (t'=z)
```

However, we avoid confusion if we use the TLA rules for priming as a convention when writing actions in HOL. In passing, we note that the variables of ordinary programs are also anonymous, in the sense that we can usually change the name of a variable throughout the program without changing the effect of executing the program.

Constants (called *rigid variables* in TLA) are formalised as variables that are not bound by λ-abstraction.

Connectives for Predicates and Actions. The boolean connectives are lifted to predicates and actions in a straightforward way. The connectives for predicates are called **pnot**, **pand**, **por**, **pimp** etc. Similarly, the connectives for actions are called **anot**, **aand**, etc. As examples, we show the theorems that define the lifted conjunctions:

$\vdash_{def}$ `p pand q = λs. p s ∧ q s`
$\vdash_{def}$ `a aand b = λs s'. a s s' ∧ b s s'`

where `s` and `s'` are states.

We also define functions that represent validity:

$\vdash_{def}$ `pvalid p = ∀s. p s`
$\vdash_{def}$ `avalid a = ∀s s'. a s s'`

2.3 Temporal Logic in HOL

We have formalised the TLA logic by semantically embedding in HOL in a straighforward way. Time is represented by natural numbers and *behaviors* have type `:num→state` where `:state` is the type of the state (in the generic case, `:state` is just a polymorphic type). Temporal formulas are represented as *temporal properties*, i.e., as boolean functions on behaviors. The following theorem defines the $\Box$-operator:

$\vdash_{def}$ `box f t = ∀i. f(λn. t(i+n))`

for temporal formula `f` and behavior `t`.

We also lift the boolean connectives to temporal formulas, giving them the names **tnot**, **tand**, etc. For example, the following theorems define lifted conjunction and lifted validity:

$\vdash_{def}$ `f tand g = λt. f t ∧ g t`
$\vdash_{def}$ `tvalid f = ∀t. f t`

where `t` is a behavior.

2.4 TLA Formulas in HOL

We have formalised TLA in HOL by directly defining the semantics of TLA. The HOL system also permits another approach, where one defines a new type corresponding to TLA formulas and then define the semantics separately. However, the syntax of TLA is such that it would be quite involved to define TLA formulas as a new type.

One consequence of our approach is that TLA formulas are a subset of the of type : `(num→state)→bool` which represents temporal properties. Thus we do not formalise the notion of "well-formed TLA formula" at all. This is justified by a pragmatic viewpoint: our aim is to mechanise TLA-reasoning about algorithms, not to reason about the properties of the TLA logic itself.

TLA Formulas as Temporal Properties. In TLA, every predicate can also be interpreted as an action or as a temporal formula. I n our formalisation this is not true. Instead, predicates, actions and temporal formulas have distinct types. We define functions that do the "lifting" from predicates to the action and temporal levels (see Fig. 1).

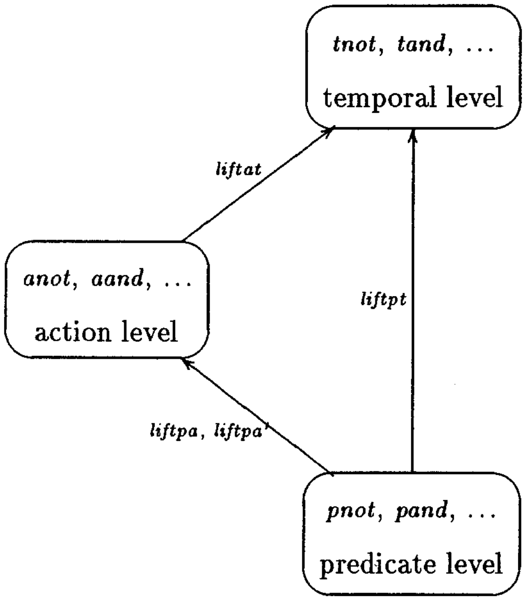

Fig. 1. Levels of the TLA logic

The functions **liftpa** and **liftpa'** that lift predicates to actions are defined as follows:

$\vdash_{def}$ `liftpa p` $=$ λ`s s'. p s`
$\vdash_{def}$ `liftpa' p` $=$ λ`s s'. p s'`

Similarly, the functions **liftpt** and **liftat** that lift predicates and actions to the temporal level are defined as follows:

$\vdash_{def}$ `liftpt p t` $=$ `p (t 0)`
$\vdash_{def}$ `liftpa a t` $=$ `a (t 0) (t 1)`

The square function $[\mathcal{A}]_f$ and the *Enabled* predicate are formalised by the following defining theorem:

$\vdash_{def}$ `square a f` $=$ `a aor (`λ`s s'. f s` $=$ `f s')`
$\vdash_{def}$ `enabled a` $=$ λ`s.` $\exists$`s'. a s s'`

where `f` is a function from states to an arbitrary type.

Finally, we consider how an algorithm is represented. We assume that we are given a predicate (the initialisation `init`), an action (`action`) and a temporal formula (the liveness part `live`). The algorithm is then described by the formula

```
(liftpt init) tand (box (liftat (square action w))) tand live
```

where `w` represents the function $\lambda(x_1, ..., x_n, z).(x_1, ..., x_n)$ which takes the global state as argument and returns the part of the state that the algorithm works on (i.e., the state with the last component removed).

An Example Algorithm. As an example we consider the algorithm described in (3). The following defining theorems show how this algorithm is formalised:

```
⊢def Init1 = λ(x,y,z). (x=0) ∧ (y=0)
⊢def M1    = λ(x,y,z)(x',y',z'). (x'=x+1) ∧ (y'=y)
⊢def M2    = λ(x,y,z)(x',y',z'). (y'=y+1) ∧ (x'=x)
⊢def w     = λ(x,y,z). (x,y)
⊢def F     = (WF M1 w) tand (WF M2 w)
⊢def Prog1 = (liftpt Init1) tand
              (box (liftat (square (M1 aor M2) w))) tand F
```

3 The TLA Logic in HOL

We will not state all the proof rules of Simple TLA. Instead, we consider two typical rules in some detail. The first rule is called *INV*1 and is used to prove invariance properties:

$$\frac{I \wedge [\mathcal{N}]_f \Rightarrow I'}{I \wedge \Box[\mathcal{N}]_f \Rightarrow \Box I}$$

This rule expresses the fact that if all program actions preserve I, then $\Box I$ holds provided I holds initially.

In HOL, the rule *INV*1 becomes

```
⊢ avalid (((liftpa I) aand (square N f)) aimp (liftpa' I))
    ⇒
    tvalid (((liftpt I) tand (box (square N f))) timp (box (liftpt I)))
```

Note how the HOL implication corresponds to the meta-implication in the rule.

The TLA rules that are used for reasoning about liveness are generally more complicated than the above rule. An example is the rule *SF*1 which permits the deduction of liveness properties from strong fairness assumptions:

$$\frac{\begin{array}{r} P \wedge [\mathcal{N}]_f \Rightarrow (P' \vee Q') \\ P \wedge \langle \mathcal{N} \rangle_f \Rightarrow Q' \\ \Box P \wedge \Box[\mathcal{N}]_f \wedge \Box F \Rightarrow \Diamond \mathit{Enabled}\langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge SF_f(\mathcal{A}) \wedge \Box F \Rightarrow (P \leadsto Q)}$$

The formalisation of this rule in HOL is quite a big expression. W hen the rule is applied, however, one never has to work with the rule as a whole. Instead, the present goal is matched against the conclusion of the rule, and three subgoals are produced, one for each assumption of the rule.

Rules of TLA, such as the above two, are represented as theorems in HOL which are proved using the definition of the semantics. At first sight the rules in HOL may look ugly, since we cannot mix predicates, actions and temporal formulas as neatly as TLA does; we have to use the lifting functions. However, this is sometimes an advantage, since we avoid the confusion which can arise when the same term is sometimes a predicate and sometimes a temporal formula. The proofs (in HOL) of the above rules are quite straighforward, using rewriting and (in the case of the liveness rules) induction.

From the basic proof rules, further rules can easily be derived. The proofs of such rules in HOL mirror the paper-and-pen proofs almost exactly.

Use of Well-founded Sets. An important rule in liveness proofs is the Lattice rule, which encodes the principle of well-founded ranking. This rule presupposes the existence of a well-founded order on some set involved in the reasoning. To support the use of this rule in HOL reasoning, we have created a separate theory of well-founded sets where the notion of well-foundedness is defined and basic properties of well-founded sets are proved.

4 Reasoning about Algorithms

Once we have proved the proof rules of Simple TLA, we can use them to reason about algorithms in HOL. We shall now give examples of two kinds of reasoning: proving that a program has a certain property and proving that one program implements another one. Formally, these amount to the same thing in TLA, since programs are formulas in the logic.

4.1 Verifying Properties of Programs

As an example, we shall show how mutual exclusion is proved for a simple algorithm. The example is taken from [6]. It is a refinement of the simple example considered earlier which incremented variables x and y indefinitely. This algorithms works on the variables x, y, pc_1, pc_2 and sem. The variables pc_1 and pc_2 are program counters, taking values $''A''$, $''B''$ and $''G''$, and sem is a semaphore.

The initial state is characterised by the predicate

$$\mathit{Init}_2 \overset{\text{def}}{=} pc_1 = \ ''A'' \wedge pc_2 = \ ''A'' \wedge x = 0 \wedge y = 0 \wedge sem = 1$$

There are six actions; two for the actual incrementing of x and y and four for handling the semaphore:

$$\begin{aligned}
\alpha_1 &\overset{\text{def}}{=} pc_1 = "A" \wedge sem>0 \wedge pc_1' = "B" \wedge sem' = sem-1 \wedge Unch(x, y, pc_2)\\
\alpha_2 &\overset{\text{def}}{=} pc_2 = "A" \wedge sem>0 \wedge pc_2' = "B" \wedge sem' = sem-1 \wedge Unch(x, y, pc_1)\\
\beta_1 &\overset{\text{def}}{=} pc_1 = "B" \wedge pc_1' = "G" \wedge x' = x+1 \wedge Unch(y, pc_2, sem)\\
\beta_2 &\overset{\text{def}}{=} pc_2 = "B" \wedge pc_2' = "G" \wedge y' = y+1 \wedge Unch(x, pc_1, sem)\\
\gamma_1 &\overset{\text{def}}{=} pc_1 = "G" \wedge pc_1' = "A" \wedge sem' = sem+1 \wedge Unch(x, y, pc_2)\\
\gamma_1 &\overset{\text{def}}{=} pc_2 = "G" \wedge pc_2' = "A" \wedge sem' = sem+1 \wedge Unch(x, y, pc_1)
\end{aligned}$$

where $Unch\ f$ is defined to mean $f' = f$. An intuitive picture of this algorithm is given in Fig. 2).

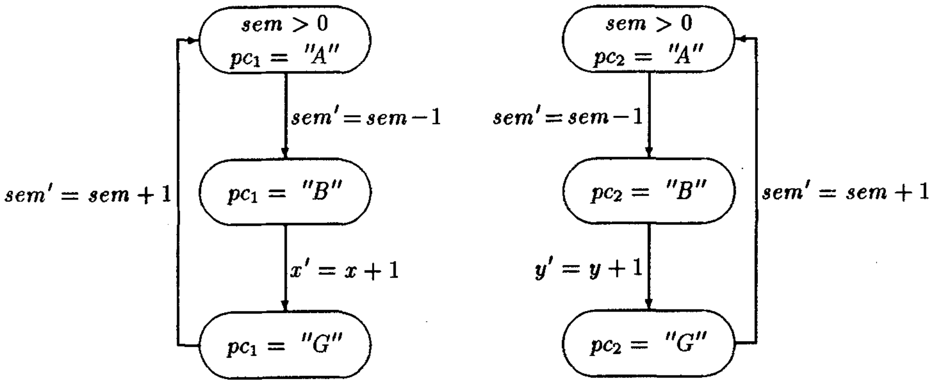

Fig. 2. Intuitive description of the algorithm Π_2

The algorithm is described formally by the formula

$$\Pi_2 \overset{\text{def}}{=} Init_2 \wedge \Box[\mathcal{N}_1 \vee \mathcal{N}_2]_w \wedge SF_w(\mathcal{N}_1) \wedge SF_w(\mathcal{N}_2)$$

where $N_1 = \alpha_1 \vee \beta_1 \vee \gamma_1$ and $N_2 = \alpha_2 \vee \beta_2 \vee \gamma_2$ and $w = (x, y, pc_1, pc_2, sem)$. In the fairness condition strong fairness is used in order to guarantee that the refined program implements the program Π_1 (cf. Sect. 4.2).

We are interested in the mutual exclusion property of this algorithm, i.e., the property that the algorithm never reaches a situation where $pc_1 = pc_2 =" B"$ holds. This is shown by proving the formula:

$$\Pi_2 \Rightarrow \Box(pc_1 = "A" \vee pc_2 = "A")$$

The translation of this algorithm into our HOL-formalisation is straightforward. The state is represented by the tuple `(x,y,pc1,pc2,sem)` which has the type `:num#num#tri#tri#num#**`. Here `:tri` is a flat type with the three elements `A`, `B` and `G`, defined by the means of the built-in type definition package of the HOL system. The possibility to define types in this way turns out to be very handy in program verification. If the program counters were defined to be, e.g., of the type `:num`, we would have needed additional arithmetic reasoning, which is tedious in HOL.

The initialisation is defined by the defining theorem

```
⊢def Init2 =
       λ(x,y,pc1,pc2,sem,z).
         (pc1=A) ∧ (pc2=A) ∧ (x=0) ∧ (y=0) ∧ (sem=1)
```

As an example of the actions, we show the theorem which defines the action β_1:

```
⊢def Beta1 =
     λ(x,y,pc1,pc2,sem,z)(x',y',pc1',pc2',sem',z').
       (pc1=B) ∧ (pc1'=G) ∧ (x'=x+1) ∧ (y'=y) ∧ (sem'=sem) ∧ (pc2'=pc2)
```

The other five actions are defined accordingly. We also define

```
⊢def N1    = Alpha1 aor Beta1 aor Gamma1
⊢def N2    = Alpha2 aor Beta2 aor Gamma2
⊢def w     = λ(x,y,pc1,pc2,sem,z).(x,y,pc1,pc2,sem)
⊢def F2    = (SF N1 w) tand (SF N2 w)
⊢def Prog2 = (liftpt Init2) tand
             (box (liftat (square (N1 aor N2) w))) tand F2
```

Now the mutual exclusion property of this algorithm can be proved in the theorem

```
⊢ tvalid
     (Prog2 timp (box (liftpt(λ(x,y,pc1,pc2,sem,z).(pc1=A) ∨ (pc2=A)))
```

This theorem is proved by means of a rule for proving invariance properties using invariants. The invariant is in this case the following:

$$sem + (pc_1 = \text{"}A\text{"} \mapsto 0|1) + (pc_2 = \text{"}A\text{"} \mapsto 0|1) \;=\; 1$$

where we use the notation $b \mapsto e|f$ for conditional expressions.

Using this invariant, the proof of the mutual exclusion property is quite straighforward. However, it involves some arithmetic which means that it is not trivial in HOL.

4.2 Proving Correctness of Implementations

The two algorithms we have considered were chosen so that Π_2 is an implementation of Π_1. To prove this, we have to prove the validity of the formula $\Pi_2 \Rightarrow \Pi_1$. This can be reduced to proving the following:

$$Init_2 \Rightarrow Init_1 \tag{4}$$
$$\Box[\mathcal{N}_1 \vee \mathcal{N}_2]_w \Rightarrow \Box[\mathcal{M}_1 \vee \mathcal{M}_2]_{(x,y)} \tag{5}$$
$$\Pi_2 \Rightarrow WF_{(x,y)}(\mathcal{M}_1) \wedge WF_{(x,y)}(\mathcal{M}_2) \tag{6}$$

where w is the state tuple of Π_2.

Note that the types of the state spaces of the two programs are as follows:

```
Prog1 : num#num#*
Prog2 : num#num#tri#tri#num# * *
```

These types match, since the type variable `:*` ("the rest of the world" for `Prog1`) can be instantiated to `:tri#tri#num#**`. In fact, this instantiation is done automatically by the HOL system.

Formulas (4–6) can be translated directly into HOL goals. The proofs of (4) and (5) are straightforward, though time-consuming, as HOL is quite slow in working with tuples.

The proof of (6) is more complicated and rather lengthy. The complication arises partly from the use of TLA's Lattice-rule. Informally this kind of proof is easy to handle, as one can refer to pictures when reasoning about simple well-founded sets, but in HOL it quickly becomes complicated. Furthermore, the formulas used for proving liveness properties are the most complex ones in the TLA proof system.

A general source of complication is the fact that the goals often must be transformed before they can be matched to the proof rules of the TLA logic. These transformations can generally be justified by appealing to some simple tautology. This means that a lot of effort goes into proving tautologies on both the predicate, action and temporal levels, and into doing transformations on the goals.

5 Conclusion

Our work shows that TLA can be represented in HOL in such a way that reasoning about algorithms can be mechanically checked. We have defined the semantics of the basic concepts of TLA and proved part of the basic proof rules of TLA as HOL theorems. Once this was done, reasoning about algorithms could be done in HOL in a way which corresponds closely to the way reasoning is carried out on paper. Thus HOL works as a proof checker for TLA reasoning.

Since we embed TLA semantically in HOL, our formalisation does not permit meta-level reasoning about the TLA logic. It is possible to formalise TLA in HOL in a way which would permit such reasoning, but such a formalisation would be much more difficult to use in practice. Melham discusses this problem in his formalisation of the pi-calculus in HOL [7].

TLA reasoning involves proofs on two levels: the temporal level and the level of actions. Proofs on the temporal level involve using the proof rules of TLA and simple temporal reasoning. Proofs on the action level involve reasoning over the datatypes that the program handles (integers, lists, etc.). We have built a theory for the first kind of reasoning; action reasoning in HOL is supported by the numerous libraries that are part of the HOL system.

Reasoning using HOL can be slow and tedious, for many reasons. A lot of effort goes into the transformation of goals into a form which matches the conclusion of some proof rule of TLA. Reasoning using well-founded sets involves a lot of straighforward but tedious proof details. Furthermore, reasoning on the action level is often arithmetic reasoning, at which HOL is notoriously inefficient. Many of these problems could be avoided if HOL could be made to support automatic proofs. In particular, we think that much of TLA reasoning could be automated if the user could supply the system with hints (e.g., invariants and definitions of well-founded orders) and if good libraries (e.g., for semi-automatic arithmetic) were added to the HOL system, for use in action level proof.

In the near future we will investigate to what extent the proofs reported in this paper can be re-used in other similar examples. This will give an indication of the degree of automation that is possible. Certainly such automation will require a substantial amount of programming in ML, the meta-language of HOL.

The definition of algorithms in HOL notation is unnecessarily difficult. This problem could be solved by designing some kind of interface which allows the user to describe algorithms in a TLA-like notation which would then be automatically translated into HOL. Such interfaces are not supported by the HOL system at present.

One important aspect of TLA is the treatment of simulation relations between programs. Simulation is proved using *refinement mappings*. In TLA this is based on existential quantification of program variables. Expressing this in HOL is left for future work.

Acknowledgements

We wish to thank the anonymous referees for their comments and helpful suggestions.

References

1. R.J.R. Back and J. von Wright. Refinement concepts formalised in higher-order logic. *Formal Aspects of Computing*, 2:247–272, 1990.
2. A.J. Camilleri. Mechanizing CSP trace theory in higher order logic. *IEEE Transactions of Software Engineering*, 16(9):993–1004, 1990.
3. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56 –68, 1940.

4. The HOL System Documentation. Cambridge, 1989.
5. L. Lamport. A temporal logic of actions. Techn. Rep. 57, DEC Systems Research Center, April 1990.
6. L. Lamport. The temporal logic of actions. Manuscript, January 1991.
7. T.F. Melham. A mechanized theory of the π-calculus in HOL. Techn. Rep. 244, University of Cambridge Computer Laboratory, January 1992.
8. J. von Wright. Mechanising the temporal logic of actions in HOL. In *Proceedings of the 1991 HOL Tutorial and Workshop.* ACM, August 1991.