A HOL Formalisation of the Temporal Logic of Actions

Thomas Långbacka

Åbo Akademi University, Department of Computer Science
Lemminkäinengatan 14–18, SF–20520 Åbo, Finland.
Email: `langback@aton.abo.fi`

Abstract. We describe an attempt to formalise the semantics of the Temporal Logic of Actions (TLA) by means of the HOL theorem prover. Special concern has been devoted to trying to formalise the rules that govern refinement mappings and data hiding in the TLA framework.

1 Introduction

The HOL system [7] has recently become more and more popular as a basis for mechanising logics and programming semantics. Examples include, but are not restricted to, Refinement calculus [3, 10], CSP [4] and UNITY [2]. This paper describes an attempt at formalising the temporal logic of actions (TLA) [8] using the HOL system.

TLA is a logic designed mainly for reasoning about concurrent algorithms. TLA is based on temporal logic, but allows refinement of specifications by ensuring (via it's syntax) that all TLA formulas are invariant under stuttering. In TLA algorithms are expressed directly as formulas in the TLA logic, rather than as programs in a separate programming language. However, reasoning about algorithms on the level of the logic involves a lot of tedious proof details. Hence, much would be gained if the reasoning could be (partly) automated using a mechanical proof assistant.

In this paper, we base ourselves on previous work reported in [9, 12]. Contrary to the work mentioned above we deal with the full TLA logic, i.e. we have made an attempt at formalising the rules that deal with refinement mappings [1] and data hiding in the TLA framework. Mechanising TLA reasoning without these provides a good base for verifying program properties but does not adequately support proofs of correctness of program development steps.

Our formalisation is done by defining the semantics of TLA directly in HOL (a so called "shallow" embedding). 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.

There has been work done on automating TLA by means of the Larch Prover [6]. This paper is not very detailed, making comparisons between that work and our work difficult. The paper does, however, present a translator from high-level TLA formulas to the syntax of the theorem prover. Such a tool would be very convenient for our HOL formalisation as well. Chou [5] formalises parts of TLA in HOL, but like in the previous work in [9, 12], refinement mappings and data hiding are not dealt with.

We assume that the reader is familiar with the HOL system. When referring to HOL-terms and interaction with the HOL system we use the syntax of HOL. To make formulas more readable, we omit type information which can be inferred from the context or is given elsewhere in the text.

2 The Temporal Logic of Actions

The Temporal Logic of Actions (TLA) is a logic for reasoning about concurrent algorithms. One of the main ideas of TLA is that algorithms are expressed directly in the logic, and not as programs in a separate programming language.

TLA is based on a simple temporal logic, with $\Box$ ("always") as the only primitive temporal operator. TLA allows refinement of algorithms by guaranteeing (through its syntax) that all well-formed TLA formulas are invariant under stuttering.

2.1 Predicates, Actions and Temporal formulas

The TLA logic has three different levels. One can express *predicates*, *actions* and *temporal formulas*. Predicates are interpreted as in ordinary predicate logic. The semantics of temporal formulas is defined by means of infinite state sequences, *behaviours*. Given some behaviour, the validity of a temporal formula is evaluated over all the states in the behaviour. Thus the meaning of a temporal formula is the set of behaviours that satisfy that formula.

An *action* is a boolean expression that relates two states (the *initial* and *final* state of that action) to each other. As an example,

$$(x' = x + 1) \wedge (y' = y)$$

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. In this case they put no restriction on the final state (or the initial state in case they appear primed as in the rule INV1 in Figure 1). Similarly, both predicates and actions can be interpreted as temporal formulas. In this case a predicate asserts something about the first state of a behaviour (or the second if primed) while an action relates the first two states of a behaviour.

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

$$[\mathcal{A}]_f \triangleq \mathcal{A} \vee (f = f')$$

where f' is the same as f but with all occurrences of program variables primed (The symbol $\triangleq$ is to be read "is defined to be equal to"). For 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 modeled. By allowing actions to appear only in the form above invariance under stuttering is achieved[1].

2.2 Expressing algorithms in TLA

An algorithm is described by a formula

$$\Pi = \mathit{Init} \wedge \Box[\mathcal{N}]_f \wedge F \tag{1}$$

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 (1). This shows how TLA permits predicates, actions and temporal formulas to be combined.

The safety part of (1) 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. Thus $\Box[\mathcal{N}]_f$ 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

$$\begin{aligned} \mathrm{WF}_f(\mathcal{A}) &\triangleq (\Box\Diamond\langle\mathcal{A}\rangle_f) \vee (\Box\Diamond\neg \mathit{Enabled}\langle\mathcal{A}\rangle_f) \\ \mathrm{SF}_f(\mathcal{A}) &\triangleq (\Box\Diamond\langle\mathcal{A}\rangle_f) \vee (\Diamond\Box\neg \mathit{Enabled}\langle\mathcal{A}\rangle_f) \end{aligned}$$

where $\langle\mathcal{A}\rangle_f \triangleq \neg[\neg\mathcal{A}]_f$ and $\mathit{Enabled}(\mathcal{A})$ is true of those states in which it is possible to perform $\mathcal{A}$. The meaning of the symbol $\Diamond$ ("eventually") is defined as $\Diamond\mathcal{A} \triangleq \neg\Box\neg\mathcal{A}$.

As a proof system, TLA has only a small set of *basic* proof rules (see [8] for all details). However, a set of *additional* rules are added for convenience. In addition to the set of pre-defined rules, temporal reasoning as well as ordinary mathematical reasoning is used. In Figure 1 we list some frequently used rules of TLA. The meaning of the symbol $\leadsto$ ("leads-to"), that appears in the LATTICE rule, is defined as $\mathcal{A} \leadsto \mathcal{B} \triangleq \Box(\mathcal{A} \Rightarrow \Diamond\mathcal{B})$.

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.

2.3 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

[1] The state function f in a formula of the form $[\mathcal{A}]_f$ has to be a tuple containing all program variables in $\mathcal{A}$ in order to guarantee invariance under stuttering.

LATTICE.

$\succ$ well-founded partial order on nonempty set S

$$\frac{F \wedge (\mathsf{c} \in \mathsf{S}) \;\Rightarrow\; (H_{\mathsf{c}} \leadsto (G \vee \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \wedge H_{\mathsf{d}}))}{F \;\Rightarrow\; ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \leadsto G)}$$

STL4.

$$\frac{F \Rightarrow G}{\Box F \Rightarrow \Box G}$$

INV1.

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

TLA2.

$$\frac{P \wedge [\mathcal{A}]_f \;\Rightarrow\; Q \wedge [\mathcal{B}]_g}{\Box P \wedge \Box[\mathcal{A}]_f \;\Rightarrow\; \Box Q \wedge \Box[\mathcal{B}]_g}$$

where

F, G, H_{c}, H_{d} are TLA formulas	P, Q, I are predicates
$\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$ are actions	f, g are state functions

Fig. 1. Some frequently used proof rules of TLA.

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. Often, however, it is assumed that variables only hold values of a certain predefined subset of the set of values (e.g. the set of integers etc.). In this case the type correctness of variables has to be proven, i.e. that a formula of the form $\Pi \Rightarrow \Box T$ holds (assuming Π is a specification of an algorithm and T is a predicate asserting that the program variables belong to the sets of values that they are supposed to).

2.4 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

$$Init_1 \triangleq (x = 0) \wedge (y = 0)$$

The incrementation is defined by the following two actions:

$$\mathcal{M}_1 \triangleq x' = x + 1 \wedge \mathit{Unch}\ y \tag{2a}$$

$$\mathcal{M}_2 \triangleq y' = y + 1 \wedge \mathit{Unch}\ x \tag{2b}$$

A predicate of the form *Unch* x means that $x' = x$. In TLA the fact that the value of program variables aren't changed has to be expressed explicitly, otherwise their value in the next state is not known.

The fairness requirement of the algorithm in 2 is weak fairness with respect to both actions, so the algorithm is described by the following formula:

$$\Pi_1 = \mathit{Init}_1 \wedge \Box[\mathcal{M}_1 \vee \mathcal{M}_2]_{(x,y)} \wedge \mathrm{WF}_{(x,y)}(\mathcal{M}_1) \wedge \mathrm{WF}_{(x,y)}(\mathcal{M}_2) \qquad (3)$$

2.5 Refinement mappings and data hiding

In TLA program variables can be hidden using existential quantification as in: $\boldsymbol{\exists} x : \Pi$ (here Π is assumed to be a specification of an algorithm in the normal sense) Informally this means that we do not constrain the value of the variable x at all, it is sufficient that there is some sequence of values such that Π holds.

Figure 2 describes the formal semantics of the existential quantifier. Given this rather complicated definition one can prove that for any TLA formula F, any variable x and any behaviours s_1 and s_2 the following holds

$$\natural s_1 = \natural s_2 \Rightarrow s_1[\![\boldsymbol{\exists}\, x : F]\!] = s_2[\![\boldsymbol{\exists}\, x : F]\!] \qquad (4)$$

stating that all existentially quantified TLA formulas are invariant under stuttering. The validity of the formula is actually independent of whether F is invariant under stuttering and the properties of the function $\natural$[2].

$$\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle =_x \langle\!\langle t_0, t_1, \ldots \rangle\!\rangle \;\triangleq\; \forall\, \mathsf{n} \in \mathsf{Nat} : \forall\, `v\text{'} \neq `x\text{'} : s_{\mathsf{n}}[\![v]\!] = t_{\mathsf{n}}[\![v]\!]$$

$$\begin{aligned}\natural\langle\!\langle s_0, s_1, s_2, \ldots \rangle\!\rangle \;\triangleq\; &\textbf{if } \forall \mathsf{n} \in \mathsf{Nat} : s_{\mathsf{n}} = s_0\\ &\quad \textbf{then } \langle\!\langle s_0, s_0, s_0, \ldots \rangle\!\rangle\\ &\quad \textbf{else } \textbf{if } s_1 = s_0 \textbf{ then } \natural\langle\!\langle s_1, s_2, s_3, \ldots \rangle\!\rangle\\ &\qquad\qquad\qquad\qquad \textbf{else } \langle\!\langle s_0 \rangle\!\rangle \circ \natural\langle\!\langle s_1, s_2, \ldots \rangle\!\rangle\end{aligned}$$

$$\sigma[\![\boldsymbol{\exists}\, x : F]\!] \;\triangleq\; \exists\, \rho, \tau \in \mathbf{St}^\infty : (\natural\sigma = \natural\rho) \wedge (\rho =_x \tau) \wedge \tau[\![F]\!]$$

where

$\circ$ denotes concatenation of sequences	σ is a behavior
$\mathbf{St}^\infty$ denotes the set of all behaviours	s, s_0, t_0, s_1, t_1, $\ldots$ are states

Fig. 2. Semantics of quantification in TLA.

Although the existential quantifier $\boldsymbol{\exists}$ clearly differs from the quantifier of predicate logic, it still obeys the laws from predicate logic (the laws can be derived using the basic rules governing quantification that are given in Figure 3). Typically, proving that an algorithm implements another means proving a

[2] Of course the question as to whether (4) actually correctly captures the notion of invariance under stuttering or not is dependant on the properties of the function $\natural$.

formula of the form $\exists y : \Pi' \Rightarrow \exists x : \Pi$. Assuming that y does not occur free in Π this formula is proven by giving a refinement mapping [1] (that we can call $\overline{x}$) that expresses x by means of the variables in Π' and then proving the formula $\Pi' \Rightarrow \overline{\Pi}$ where $\overline{\Pi}$ stands for Π with every occurrence of x replaced by $\overline{x}$. After this we simply apply rule E1 to deduce $\Pi' \Rightarrow \exists x : \Pi$ and then by applying rule E2 we are done.

E1.
$\vdash F(g/x) \Rightarrow \exists\, x : F$

E2.

$$\frac{F \;\Rightarrow\; G \qquad x \text{ does not occur free in } G}{(\exists\, x : F) \;\Rightarrow\; G}$$

where
x is a variable — F, G are TLA formulas
g is a state function

Fig. 3. Rules for quantification in TLA.

3 TLA in HOL

3.1 States, Predicates and Actions in HOL

In our formalisation, we assume that every variable has a well-defined type. 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. Thus, this final component ("the rest of the universe") can be instantiated to any tuple. This is needed to allow refined versions of the algorithm to add new variables to the state if needed. Our choice of representation of states was inspired by the experiences gained by the developers of a HOL formalisation of the Refinement Calculus in HOL (see [3] vs. [10]).

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#'s` (`'s` is a type variable).

Predicates and actions are formalised as boolean expressions with lambda-bound program variables. For example, in the above mentioned state space, the action $\mathcal{M}_1$ of (3) 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

Note that the variables are anonymous, in the sense that the previous action is equivalently expressed by

λ(a,z,x)(c,t$'$,z$'$). (c=a+1) $\wedge$ (t$'$=z)

However, we avoid confusion if we use the TLA rules for priming as a convention when writing actions in HOL.

Constants (called *rigid variables* in TLA) are formalised as variables that are not bound by λ-abstraction. This representation directly mirrors the meaning of rigid variables in TLA, which are not constants in the normal meaning of the word, but rather variables whose value is fixed but unknown.

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 $\wedge$ q s
$\vdash_{def}$ a aand b = λs s$'$. a s s$'$ $\wedge$ b s s$'$

where `s` and `s'` are states. We also define functions that represent validity:

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

3.2 Temporal Logic in HOL

We have formalised the TLA logic by semantically embedding in HOL in a straightforward way. Behaviours have type `:num`$\rightarrow$`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 behaviours. The following theorem defines the $\Box$-operator:

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

for temporal formula `f` and behaviour `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 $\wedge$ g t
$\vdash_{def}$ tvalid f = $\forall$t. f t

where `t` is a behaviour.

3.3 TLA Formulas in HOL

We have formalised TLA in HOL by directly defining the semantics of TLA. HOL 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 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.

In TLA, every predicate can also be interpreted as an action or as a temporal formula. In 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. 4).

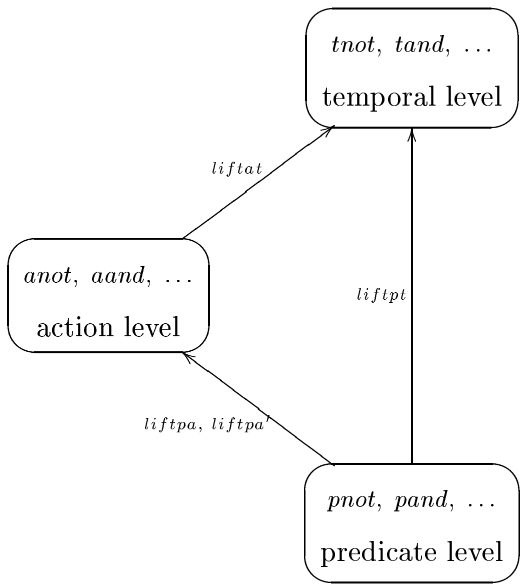

Fig. 4. Levels of the TLA logic

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:

```
⊢def liftpt p t = p (t 0)
⊢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 theorems:

```
⊢def square a f = a aor (λs s'. f s = f s')
⊢def enabled a = λs. ∃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).

3.4 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
```

Here `WF` stands for the HOL equivalent of weak fairness in TLA.

4 The Simple TLA Logic in HOL

The proof rules of the simple TLA logic (TLA without refinement mappings and data hiding) are represented as theorems in HOL which are proved using the definition of the semantics. We will not show how all of the rules are formulated in HOL. Instead, we describe the rules from Figure 1. Below we list the HOL theorems representing the rules INV1, STL4 and TLA2. This rule is frequently used when proving invariance properties of programs. It falls into the category of additional rules, i.e. rules that can be proven from the basic ones.

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

```
⊢ p pimplies q ⇒  box (liftpt p) timplies box (liftpt q)

⊢ (liftpa p aand square a f) aimplies (liftpa q aand square b g) ⇒
  (box (liftpt p) tand box (liftat (square a f))) timplies
  (box (liftpt q) tand box (liftat (square b g))))
```

The `pimplies` function above is shorthand for `pvalid(p pimp q)` and likewise for the `timplies` function. Note how the HOL implication corresponds to the meta-implication in the rule. The proofs (in HOL) of the above rules are quite straightforward, using rewriting and other basic HOL proof methods.

An important (basic) rule in liveness proofs is the LATTICE rule (see Figure 1), 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 allow the formulation of this rule in HOL reasoning, we have used a separate theory of well-founded sets[3] where the notion of well-foundedness is defined and basic properties of well-founded sets are proved. In HOL the theorem representing the LATTICE rule is as follows:

```
⊢ wellf po
  ⇒  (∀c.f timplies ((h c) leads_to (g tor (λt.∃d.po d c ∧ h d t))))
  ⇒  f timplies ((λt.∃c. h c t) leads_to g)
```

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.

In the revised version of TLA [8] some of the basic as well as additional rules have changed. We have not yet proved all of the changed additional rules.

5 Refinement mappings and data hiding in HOL

We have formalised the notion of data hiding in a way resembling the method used in [11] for defining data refinement in the refinement calculus. Thus, the state space is extended to consist of two parts: a *hidden* and a *visible* part. A behaviour over such an extended state space is a function of type `num->('h#'s)` in the generic case.

The way the quantifier $\boldsymbol{\exists}$ is defined in [8] it actually obeys the same laws as the ordinary existential quantifier of predicate logic. The way we define quantification below may not be as general, but we feel that it (together with our formalisation of the rules E1 and E2) is sufficient (with suitable type instantiations in the rules) for proving refinement steps involving refinement mappings.

[3] Due to Jockum von Wright

5.1 Basic definitions

Using the above representation we define the existential quantifier $\exists$ as a function `ex` according to the following:

```
⊢ ∀f s. ex f s = (∃t x. (flat s = flat t) ∧ f(λn. x n,t n))
```

where `f` is a formula over an extended state space and `s` is a behaviour over a non-extended state space. In this representation multiple quantification can be expressed by instantiating the hidden component as a tuple. The function `flat` (representing the function ♮ from Figure 2) is defined as follows:

```
⊢ ∀b. flat b =
   (λn.(∃m. m - stut b m = n) → (b (εm. m - stut b m = n)) | (b n))
```

Here `b` is a behaviour over a simple state space and `stut` is a recursively defined function that given a behaviour `b` and a variable `m` of type `num` returns the number of stuttering steps that the behaviour has performed during the time period `0..m`.

5.2 Proof rules

We proceed by formulating the HOL equivalents of the proof rules E1 and E2 from Figure 3 and prove them to be HOL theorems. Given the definitions above the proof rule E1 becomes:

```
⊢ ∀f p g. f (λn. g (p n),SND (p n)) ⇒ ex f (λn. SND (p n))
```

where `f` is a formula over an extended state space, `p` is a behaviour over an extended state space and `g` is a function of type `('hh#'a)->'h` (a refinement mapping). The rule E2 has the following format:

```
⊢ ∀f g. ((∀s. f s ⇒ g (λn. SND (s n))) ∧
          (∀p p′. (flat p = flat p′) ⇒ (g p = g p′))) ⇒
         ∀t. (ex f t ⇒ g t)
```

Here we have to add the second conjunct (on the left hand side of the meta-implication of the rule) in order to get a sound rule. In the rule `f` is a formula over an extended state space, `g` is a formula over a simple state space, `s` is a behaviour over an extended state space and `t`, `p`, `p'` are behaviours over a simple state space. The second conjunct asserts the the formula `g` is invariant under stuttering. This requirement follows from the fact that our definition of the semantics of TLA allows defining formulas that are not invariant under stuttering (however, there is an implicit assumption that only formulas allowed by the syntax of TLA are defined). However, if the formula `g` in the rule is of the form `ex h` (as it usually is) then invariance under stuttering is trivially true (as shown in section 2 in equation (4)). If the formula `g` is not existentially quantified we need a function "lifting" it to the type of extended behaviours. We also actually need to prove invariance under stuttering.

The way these rules are used is, in general, as follows. Suppose we want to prove that a specification `ex g` is correctly implemented by `ex f`. Then we need to prove the following HOL theorem:

```
⊢ ∀t. (ex f t ⇒  ex g t)
```

Here `f` and `g` are formulas over extended state spaces (they may be the same but are usually different). In order to prove this we actually prove the following theorem:

```
⊢ f timplies g′
```

where `g'` stands for the original formula `g` with the hidden component expressed by means of an refinement mapping of the type required by rule E1. The main work in proving a refinement step is done in proving the above theorem. It is of course essential that the theorem that is to be proven is actually on the level our temporal logic (as it is above) so that the proof rules can be used (in the same way as they would be used in a paper and pen proof).

By rewriting the above theorem using the semantical definitions it is shown that the following is a theorem:

```
⊢ ∀s. f s ⇒  ex g (λn. SND (s n)))
```

after which we are done since both premises of rule E2 have been established.

5.3 Applying the rules to a simple example

To show how to use the rules above in practice we consider a simple example. We define two algorithms Φ and Ψ as follows:

$$
\begin{aligned}
Init_1 &\triangleq (x, i) = (0, 0) \\
Init_2 &\triangleq (x, y, i, j) = (0, 0, 0, 0) \\
\mathcal{A}1 &\triangleq (x', i') = (x + 1, i + 1) \\
\mathcal{A}2 &\triangleq (x', y', i', j') = (x + 1, y + 2, i + 1, j + 2) \\
\Phi &\triangleq Init_1 \wedge \Box[\mathcal{A}1]_{(x,i)} \\
\Psi &\triangleq Init_2 \wedge \Box[\mathcal{A}2]_{(x,y,i,j)}
\end{aligned}
$$

What we want to prove is the validity of the formula $\exists x : \exists y : \Psi \Rightarrow \exists x : \Phi$ (i.e. that $\exists x : \exists y : \Psi$ implements $\exists x : \Phi$. The algorithms are expressed in HOL as follows:

```
⊢def Init1 = λ(x,(i,z)).(x=0) ∧ (i=0)
⊢def A1    = λ(x,(i,z))(x',(i',z')).(x'=x+1) ∧ (i'=i+1)
⊢def v     = λ(x,(i,z)).(x,i)
⊢def Prog1 = (liftpt Init1) tand (box (liftat (square A1 v)))
⊢def Init2 = λ((x,y),(i,z)).(x=0) ∧ (i=0)
⊢def A2    = λ((x,y),(i,j,z))((x',y'),(i',j',z')).
                 (x'=x+1) ∧ (y'=y+2) ∧ (i'=i+1) ∧ (j'=j+1)
⊢def w     = λ((x,y),(i,j,z)).((x,y),(i,j))
⊢def Prog2 = (liftpt Init2) tand (box (liftat (square A2 w)))
```

and the theorem we want to prove in order to establish that Prog2 correctly implements Prog1 is

```
⊢ ∀s. ex Prog2 s ⇒  ex Prog1 s
```

Using the same reasoning as above this theorem is proven by proving the validity of the following formula:

```
⊢ Prog2 timplies λs.Prog1 (λn.(FST(FST(s n)),SND(s n)))
```

A proof of a theorem as the one above typically follows the same proof outline as pen and paper proofs in TLA. However, a lot of effort goes into transforming goals so that they match the rules. Typically this means proving lots of simple tautologies on the predicate, action as well as temporal levels.

6 Conclusions

We have described an attempt at formalising TLA in HOL with the purpose of allowing mechanised reasoning of (concurrent) algorithms. Especially, we have formalised the rules that govern data hiding and refinement mappings which was not considered in the earlier work described in [9, 12].

There are however many ways in which the work presented here could be improved. Focusing specifically on the part concerning the formalisation of data hiding and refinement mappings, it needs to be tested with more realistic examples. Also there is a need for a proof procedure for proving invariance under stuttering (for formulas that follow the TLA syntax), to simplify the use of rule E2.

More generally, all the additional rules that have changed in the revised version of TLA [8], and haven't yet been proved, should be proved. These rules, that deal with fairness (which is the most difficult part in TLA reasoning), are essential in simplifying proofs of refinement.

An interface of the type described in [6], translating some higher level representation of the TLA syntax to the somewhat complicated one of HOL (as well as handling the grouping of program variables into tuples etc.), would be a great help. Also any amount of automation that could be added to the general level of TLA reasoning (in HOL) would certainly be of use.

Acknowledgements

Thanks are due to Jockum von Wright for all his help and advice. I also want to thank anonymous referees for their comments and helpful suggestions. Finally I want to thank Michael Butler for his comments on the final version of this paper.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark, Lyngby, 1992.
3. R. J. R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2:247–272, 1990.
4. A. Camillieri. Mechanizing CSP trace theory in Higher Order Logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.
5. C-T Chou. Mechanical verification of distributed algorithms in higher-order logic. In these proceedings.
6. U. Engberg, P. Groenning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification – Fourth International Workshop. CAV '92. Montreal. Canada. June 29 – July 1. 1992*, volume 663 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.
7. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.
8. L. Lamport. The temporal logic of actions. Research Report 79, DEC, Systems Research Center, December 1991. A revised version of the paper will appear in *ACM Transactions on Programming Languages and Systems*.
9. J. von Wright. Mechanising the temporal logic of actions in HOL. In *Proceedings of the 1991 HOL Tutorial and Workshop*, August 1991.
10. J. von Wright. Program refinement by theorem prover. In *BCS FACS Sixth Refinement Workshop – Theory and Practise of Formal Software Development. 5th – 7th January, City University, London, UK.*, 1994.
11. J. von Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanising some advanced refinement concepts. *Formal Methods in System Design*, 3:49–81, 1993.
12. J. von Wright and T. Långbacka. Using a theorem prover for reasoning about concurrent algortihms. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification – Fourth International Workshop. CAV '92. Montreal. Canada. June 29 – July 1. 1992*, volume 663 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.