Yet Another Encoding of TLA in Isabelle

Stephan Merz

Institut für Informatik, Universität München, Germany
http://www4.informatik.tu-muenchen.de/~merz/

Abstract

This note describes a new encoding of Lamport's Temporal Logic of Actions in the generic theorem prover Isabelle. It is based on a simple and general technique that exploits the higher-order features of Isabelle's meta-language to obtain a sound embedding of modal logics. The technique does not seem to have been used in previous formalizations of similar logics; its main advantage is that it seamlessly integrates existing facilities and techniques of Isabelle/HOL, such as inductive datatypes and automated proof strategies.

1 Background

The existence of powerful, highly automated, integrated, and extensible proof tools is an essential prerequisite for the successful practical application of formal methods. It is attractive to base such tools on standard logical frameworks [6, 13, 14, 15] that provide the necessary proof infrastructure. This note describes the design of a new embedding of Lamport's Temporal Logic of Actions [10] in the higher-order logic of the interactive theorem prover Isabelle.

Representations of object logics in logical frameworks are traditionally classified as "deep" or "shallow" embeddings. In a deep embedding, the syntax and semantics of the object logic are fully formalized in the meta-logic. For a temporal logic, this would mean to give an inductive definition of formulas and to define concepts such as state, state sequence, temporal succession, and the satisfaction relation. One can then talk about the logic not only at the object level, but also at the meta-level, and in particular prove the soundness and completeness of a proof system for the object logic. On the other hand, the advantages of deep embeddings may not be as relevant if one is mainly interested in applying an object logic as a workhorse of a formal method, where the user should not have to (or be allowed to) resort to semantic reasoning about specifications. In several cases, it has been felt that the definitional apparatus necessary to represent the object logic makes the representation rather cumbersome to apply in practice. Our embedding is therefore axiomatic and (mostly) shallow. When in doubt, we have opted for pragmatic solutions that appeared easy to apply in practice, but certainly without invalidating the correctness of the representation. The meta-theoretic justification of the axioms that underly our embedding is given in a companion note [12], which proves the soundness and completeness of the propositional part of the proof system. (It is well-known that full first-order temporal logic cannot be finitely axiomatized.)

The main problem with syntactic embeddings of modal or temporal logics in standard logical frameworks is to keep track of different modal contexts in a derivation. Proof-theoretically, the problem appears as a failure of the deduction theorem. In particular, while all (normal) modal logics contain the rule $A \vdash \Box A$ of necessitation, the formula $A \Rightarrow \Box A$ is not valid in general. It is therefore necessary to distinguish between object-level implication and entailment, which precludes a naïve use of natural-deduction style reasoning, Isabelle's preferred proof format. For example, Isabelle's standard distribution comes with a formalization of several modal logics based on first-order sequent calculus. Unfortunately, this formalization requires a duplication of much of the built-in functionality of Isabelle's meta-level implication, which makes it cumbersome to use, and it does not support the use of Isabelle's advanced tactics for automated reasoning.

In our TLA embedding, we adopt a very simple solution that keeps track of modal contexts with the help of Skolem constants, exploiting the parameter mechanism that Isabelle provides anyway. The idea is to represent formulas of "intensional" logics as functions of type $\omega \to \texttt{bool}$, where ω is a type of "possible worlds". Concrete possible-world types are defined in client theories, and can be given as much or as little structure as needed. In contrast to a semantic embedding, these types may simply be opaque types; their only purpose is to provide Skolem constants to distinguish between different contexts. In fact, our representation can be characterized as an abstraction of a semantic embedding that hides the semantics of states and state sequences. We use this idea for both temporal and action formulas of TLA: this approach simplifies the definition of TLA's concepts of primed state predicates and enabledness of actions.

Our system is based on Isabelle's representation of higher-order logic (Isabelle/HOL), which is among the best-developed object logics in Isabelle and contains many standard mathematical theories, is supported by automated proof procedures, and provides packages to define and reason about inductively defined data types, which are very useful for system specifications. For example, the principle of well-founded orderings, which is central to many liveness proofs in temporal logics, can be justified from basic axioms for temporal logic and the existing formalization of well-founded relations in Isabelle/HOL.

The outline of this note is as follows: Section 2 gives a more detailed account of the embedding of "intensional" logics that underlies our representation of TLA. Sections 3 and 4 describe the representations of the action and temporal levels of TLA. Section 5 discusses higher-level tactics that we have defined or instantiated, section 6 presents a trivial example that conveys the basic "look and feel" of our prover. (Somewhat larger examples are contained in the Internet distribution.) Finally, section 7 draws some conclusions from our experience with the embedding and compares it to existing TLA provers. We assume the reader to be familiar with TLA. Some acquaintance with Isabelle will be helpful in understanding the technicalities.

2 Representation of intensional logics

2.1 Theory Intensional

Isabelle represents subgoals during a proof as natural deduction-like sequents of the form

$$\texttt{!!}x_1 \ldots x_m.\quad \texttt{[|}\ \Phi_1\texttt{;}\ldots\texttt{;}\Phi_n\ \texttt{|]}\ \texttt{==>}\ \Phi$$

The parameters x_i are Skolem constants that accumulate in the course of the proof, Φ_1, …, Φ_n, and Φ are meta-level formulas (of type `prop`). The symbols `!!` and `==>` represent meta-level quantification and implication. Informally, the sequent above can be understood as asserting that the validity of Φ in a class of interpretations of interest follows from the validity of the formulas Φ_1, …, Φ_n over the same class, for every (type-correct) instantiation of the parameters x_1, …, x_m. However, during proofs in Isabelle/HOL it is rarely necessary to think in terms of validity, thanks to the deduction theorem of higher-order logic, which ensures that object-level implication and meta-level entailment can be identified. In fact, even the coercion from the object-level type `bool` of formulas to the meta-level type `prop` of propositions is left implicit in the syntax.

This identification is no longer possible in modal logics that do not validate the deduction theorem. We have already indicated in section 1 that our way around this problem is to introduce a "lifted", functional type of formulas. Figure 1 contains the essential parts of the Isabelle theory file `Intensional.thy` that defines intensional logics. The theory first declares a new type class `world` whose element types are intended to represent types of possible worlds. Concrete types of class `world` will be furnished by client theories; all subsequent definitions are polymorphic in a type parameter of class `world`. Intensional formulas have types of the form $\omega \rightarrow$ `bool` where ω is a type of class `world`.

The `consts` section of the theory file declares the basic syntax of "intensional" logics. The first constant, `TrueInt` is a type-conversion function that turns an intensional formula into a meta-level Isabelle proposition, and is analogous to the HOL function `Trueprop`. In particular, `TrueInt` also has empty syntax and need never be explicitly written by the user. Intuitively, the proposition `TrueInt(F)` asserts that the intensional formula `F` is valid over all interpretations of interest.

! Since both `Trueprop` and `TrueInt` have empty syntax, the grammar becomes ambiguous, although the ambiguity can always be resolved by the type checker. As explained in the Isabelle manual, the variable `Syntax.ambiguity_level` should be set to a suitable value to suppress warning messages generated by the parser.

Function `holdsAt` evaluates an intensional formula `F` at a world `w`, which can be conveniently written as `w |= F`. Since an intensional formula is a function from worlds to booleans, evaluation simply means function application, as can be seen from the (parsing) translation for `holdsAt`, which is defined in the `translations` section of the theory file.

The next group of functions concern the lifting of base-level constants and functions to the intensional level. Function `con` (with external representation `#`) lifts base constants to intensional terms. For example, `#True` is the lifted

```
Intensional  =  Prod +

classes
    world < logic

types
    'w form = "'w => bool"

consts
  TrueInt  :: "('w::world form) => prop"                    ("(_)" 5)
  holdsAt  :: "['w::world, 'w form] => bool"                ("(_ |= _)" [100,9] 8)

  con      :: "'a => ('w::world => 'a)"                     ("(#_)" [100] 99)
  lift     :: "['a => 'b, 'w::world => 'a] => ('w => 'b)"  ("_[_]")
  (* ... *)

  (* Lifted infix functions *)
  IntEqu   :: "['w::world => 'a, 'w => 'a] => 'w form"     ("(_ .= _)"  [50,51] 50)
  IntImp   :: "[('w::world) form, 'w form] => 'w form"     ("(_ .-> _)" [26,25] 25)
  (* ... *)

  (* "Rigid" quantification *)
  RAll     :: "('a => 'w::world form) => 'w form"          (binder "RALL " 10)
  REx      :: "('a => 'w::world form) => 'w form"          (binder "REX " 10)

syntax
  "@tupleInt"    :: "args => ('a * 'b, 'w) term"  ("(1{[_]})")

translations
  "u .= v"   == "op =[u,v]"
  "A .-> B"  == "op -->[A,B]"

  "{[x,y,z]}"   == "{[x, {[y,z]} ]}"
  "{[x,y]}"     == "Pair [x, y]"
  "{[x]}"       => "x"
  (* ... *)

rules
  inteq_reflection   "(x .= y) ==> (x == y)"
  int_valid   "TrueInt(A) == (!!w. w |= A)"
  unl_con     "(#c)(w) == c"
  unl_lift    "(f[x])(w) == f (x w)"
  unl_lift2   "(f[x,y])(w) == f (x w) (y w)"
  unl_lift3   "(f[x, y, z])(w) == f (x w) (y w) (z w)"
  unl_Rall    "(w |= RALL x. A(x)) == ALL x. (w |= A(x))"
  unl_Rex     "(w |= REX x. A(x))  == EX x. (w |= A(x))"
end
```

Figure 1: An Isabelle theory of intensional logics

boolean `True`, as an intensional formula. Of course, such type-conversion functions are always left implicit in textbook presentations of modal logics. In principle, it would often be possible to leave them implicit in Isabelle as well and let the parser insert the necessary liftings. But doing so would result in a highly ambiguous grammar and would slow down the parser. Rule `con_unl` in the `rules` section of the theory file asserts that a lifted constant evaluates to the base constant in every world. Observe that we implicitly assume that domains are independent of possible worlds (the so-called "constant-domain assumption").

A similar lifting of functions is performed by the function `lift` and its variants for binary and ternary functions (omitted in figure 1). The external syntax uses square brackets for the application of lifted functions. Note that lifted functions expect as arguments intensional terms; for example, `Suc[#0]` denotes an application of the lifted successor function to the lifted zero constant, whereas `#Suc` is a binary function that expects as arguments a world and a base term. As a convenience, theory `Intensional` defines lifted infix functions for many infix functions that are predefined in Isabelle/HOL, using a dot notation for the external representation. Figure 1 shows two examples for such lifted infix functions, lifted equality and lifted implication. These auxiliary functions are immediately reduced to the primitive form by the translations shown in the `translations` section of the theory file. The syntax for forming tuples of lifted terms uses brackets of the form `{[...]}`.

Finally, theory `Intensional` defines two quantifiers that correspond to rigid universal and existential quantification over base values. (Flexible quantification over state variables for temporal TLA formulas will be discussed in section 4.)

It remains to examine the axioms and rules of theory `Intensional` (recall that our embedding is an axiomatic extension of Isabelle/HOL).

Rule `inteq_reflection` says that if `x .= y` is valid, that is, if two intensional terms `x` and `y` are equal in al worlds, then they may be substituted for each other (the symbol `==` denotes Isabelle's meta-equality). This can be useful in proofs by rewriting. Observe that the left-hand side is surrounded by an implicit application of `TrueInt`; obviously, the validity of the asserted equality is essential for this rule to be sound.

Rule `int_valid` defines formula `A` to be valid iff it holds of all worlds. The remaining axioms concern the unlifting of lifted expressions; they are purely definitional and straightforward.

The file `Intensional.ML`, which is automatically read by Isabelle after processing the definitions and rules of the theory file `Intensional.thy`, proves the rules

```
intI  (!!w. w |= A) ==> A                    intD  A ==> w |= A
```

from the definition `int_valid`, which represent basic introduction and destruction rules for intensional validity. Besides, it adds a number of basic intensional simplifications to the default set of simplifications, modifies the simplifier to accept intensional equalities of the form `x .= y` as simplification rules, and

Function	Argument	Result
`int_unlift`	any theorem	unlifting, basic rewriting
`int_mp`	`F .-> G`	`w |= F  ==>  w |= G`
`int_impE`	`F .-> G`	`[| w |= F; w |= G ==> R |] ==> R`
`int_conjmp`	`F .& G .-> H`	`[| w |= F; w |= G |] ==> w |= H`
`int_conjimpE`	`F .& G .-> H`	`[| w |= F; w |= G; (w |= H ==> R) |] ==> R`
`int_rewrite`	`F .= G`	`F == G`

Figure 2: Functions to unlift intensional formulas

provides a number of useful unlifting functions for intensional formulas, summarized in figure 2.

2.2 Intensional proofs by unlifting

Let us now illustrate our theory of "intensional" logics by way of a simple example proof, which shows the typical proof strategy of our embedding. Specifically, we want to prove a chain rule

$$A \Rightarrow B, B \Rightarrow C \;\vdash\; A \Rightarrow C$$

for intensional formulas A, B, and C.

After loading theory `Intensional`, we input the following proof goal:

```
val prems = goal Intensional.thy "[| A .-> B; B .-> C |] ==> A .-> C";
   Level 0
    1. A .-> C
```

Isabelle echoes[1] the input goal as the first subgoal to be proven and records the binding of the premises to the identifier `prems`. We introduce the premises into the subgoal:

```
by (cut_facts_tac prems 1);
   Level 1
    1. [| A .-> B; B .-> C |] ==> A .-> C
```

Since the goal is an intensional formula, we use the introduction rule `intI`.

```
by (rtac intI 1);
   Level 2
    1. !!w. [| A .-> B; B .-> C |] ==> w |= (A .-> C)
```

Resolution has introduced a new parameter `w`, and we are required to prove that $A \Rightarrow C$ holds at "world" `w`. The next obvious step is to use the rule `intD` as a destruction rule on the sequent's premises.

```
by (REPEAT (dtac intD 1));
   Level 3
    1. !!w. [| (?w3 w) |= (A .-> B); (?w4 w) |= (B .-> C) |]
              ==> w |= (A .-> C)
```

[1] The transcript of the Isabelle session has been slightly edited.

The rule has introduced two unknowns `?w3` and `?w4`; Isabelle records that any later instantiation of these unknowns may depend on world `w`.

```
by (rewtac unl_lift2);
   Level 4
    1. !!w. [| A (?w3 w) --> B (?w3 w); B (?w4 w) --> C (?w4 w) |]
            ==> A w --> C w
```

Rewriting has eliminated the lifted ("intensional") connectives, and we are left with a base sequent, which can be solved using Isabelle/HOL's predefined tactic for propositional reasoning.

```
by (fast_tac prop_cs 1);
   Level 5
   No subgoals!
```

Although this is a trivial example, it shows our basic strategy for proving intensional formulas: Skolem constants that represent "worlds" are introduced to unlift the original goal. The resulting formulas can then be handled in the standard way using HOL proof rules. Although this approach may seem disappointing to logicians who would prefer a more elaborate high-level handling of modal logics, its advantage is that it requires only a small extension of Isabelle/HOL and it is very easy to automate.

It is worth noting that our derivation has been "polymorphic" over all intensional logics. Isabelle records this polymorphism, as the following transcript shows:

```
val chainW = result();
   val chainW =
       "[| (?A::?'w::world => bool) .-> (?B::?'w::world => bool);
           ?B .-> (?C::?'w::world => bool) |] ==> ?A .-> ?C" : thm
```

3 The action level of TLA

One characteristic feature of TLA is that its temporal formulas are defined on top of a layer of actions (or transition formulas), which are evaluated over pairs of states. This two-tiered definition of formulas is reflected in our embedding. We now describe the machinery used to represent the state and transition level. Section 4 documents the representation of the temporal level of TLA.

3.1 States and state variables

Theory `Stfun`, shown in figure 3, declares a type `state` and a monadic type constructor `stfun` to represent (typed) state functions. In particular, state predicates are boolean state functions. It may come as a surprise that we do not use theory `Intensional` to define a logic of state formulas. The reason is simply pragmatic: an extra level of state formulas would require more conversion functions between the state and the transition levels and make the syntax (even) heavier.

It is often necessary to identify the free state variables of a TLA specification, which represent state components of a system. (In particular, this information

```
Stfun  =  Prod +

types
    state
    'a stfun = "state => 'a"
    stpred   = "bool stfun"

arities
    state :: term

consts
  base_var  :: "'a stfun => bool"
  pairSF    :: "['a stfun, 'b stfun] => ('a * 'b) stfun"

syntax
  "@tupleSF"      :: "args => ('a * 'b) stfun"  ("(1<_>)")

translations
  "<x,y,z>"   == "<x, <y,z> >"
  "<x,y>"     == "pairSF x y"
  "<x>"       => "x"

rules
  pairSF_def  "<v,w>(s) = (v(s),w(s))"
  base_var    "base_var v     ==>  EX u. v u = c"
  base_pair   "base_var <v,w>  =   (base_var v & base_var w)"
end
```

Figure 3: A theory of state functions

is important to reason about the enabledness of actions.) Theory `Stfun` defines the predicate `base_var` that serves to identify these "base" variables to the prover. The user who defines a TLA specification should include a clause that states which state functions are the base variables. We have chosen not to distinguish (base) state variables and state functions at the type level, again because such a distinction would make the syntax heavier. Another possible solution based on lexical features of identifiers has been used in the representation of Hoare logic that comes with the standard Isabelle distribution, but it appears even more ad-hoc and less transparent to the user.

The remaining constructs of theory `Stfun` support the tupling of state functions, which is frequently used in TLA specifications. Basically, we only need to lift pairing of base types as defined in HOL's standard extension `Prod` to state functions; this is expressed by the rewrite rule `pairSF_def`. We also define some translations that simplify the external representation of tuples so that we may write, for instance, `<x,y,z>` instead of `<x,<y,z>>`.

We cannot *define* what it means for a state function to be a "base" variable, because we do not have a semantic representations of states in our theory. Two relevant properties are expressed by `base_var` and `base_pair`: The first axiom asserts that if `v` is a base variable, then for any value `c` (of correct type) we may find a state `u` that assigns `c` to `v`. The second axiom states that that a pair `v,w` of state functions can be regarded as a base variable iff both `v` and `w` are base variables. The "$\Leftarrow$" implication of `base_pair` would not be provable even if we replaced the meta-implication in `base_var` by meta-equality, because we could not prove that there exists a state that assigns suitable values to both `v` and `w`. (This indicates that the concept of "base" variables cannot be defined in our framework.)

3.2 Representing actions

Lamport argues in [10] that "the validity of actions [...] is proved by ordinary mathematical reasoning". One might therefore expect to use plain Isabelle/HOL to reason about the action level. However, in a marginal note Lamport also identifies a problem with such a naïve approach. Specifically, TLA action formulas contain primed and unprimed state variables, which have to be treated as distinct variables at the action level. The problem comes from a general "macro" defined on state functions, also denoted by $'$, that replaces all unprimed variables in a state function by their primed counterparts. Isabelle's macro facility is restricted to user input or output, not for internal use in logical theories, probably because the interaction of functions and macros would be somewhat obscure, and it can therefore not be used to define this syntactic operator. However, priming cannot be defined as a function on ordinary, first-order formulas, because it is not referentially transparent; for example, it is unsound to deduce $x' = 0$ from the formula $x = 0$ and the fact that $0'$ is 0. All existing TLA provers have been very careful to avoid substitutions in the scope of primed predicates.

Looking at the semantics of TLA, we may view the action level as a simple intensional logic, whose worlds are pairs of states. Figure 4 shows the theory `Action`, which is based on our generic theory of intensional logics described in section 2. Theory `Action` first defines a type `state2` of class `world` that represents pairs of states and types for transition functions and actions (boolean transition functions; the type constructor `form` was defined in theory `Intensional`). We have not defined type `state2` as a pair of states because this would require to overload the product type constructor to generate a `world` type from a pair of `term` types, but this technicality is hidden from the casual user of our system by the auxiliary function `mkstate2` and the rule `state2_ext`, which effectively asserts that every object of type `state2` is generated by `mkstate2`.

! Do not use the basic intensional introduction and elimination rules `intI` and `intD` for action-level proofs: there are no useful rewrites for formulas that involve type `state2`. Instead, use the derived rules `actionI` and `actionD` shown in figure 5.

Theory `Action` defines type conversion functions `before` and `after` that turn state variables into transition functions. If `v` is a state function, then `$v` corresponds to an unprimed occurrence of `v` in an action formula, whereas `v$` corresponds to a primed occurrence of `v`. Axioms `unl_before` and `unl_after` show how primed and unprimed state functions are evaluated in a pair of states. The action `unchanged f` represents inaction with respect to state function `f`; it has the same effect as the action `f$ .= $f`.

```
Action  =  Intensional + Stfun +

types
    state2      (* intention: pair of states *)
    'a trfct =  "state2 => 'a"
    action   =  "state2 form"

arities
    state2 :: world

consts
  mkstate2      :: "[state,state] => state2"  ("([[_,_]])")

  before        :: "'a stfun => 'a trfct"            ("($_)"  [100] 99)
  after         :: "'a stfun => 'a trfct"            ("(_$)"  [100] 99)
  unchanged     :: "'a stfun => action"

  prime         :: "'a trfct => 'a trfct"            ("(_`)" [90] 89)

  SqAct         :: "[action, 'a stfun] => action"    ("([_]'_(_))" [0,60] 59)
  AnAct         :: "[action, 'a stfun] => action"    ("(<_>'_(_))" [0,60] 59)
  Enabled       :: "action => stpred"

rules
  state2_ext    "(!!s t. [[s,t]] |= (A::action)) ==> (st::state2) |= A"

  unl_before    "($v) [[s,t]] == v s"
  unl_after     "(v$) [[s,t]] == v t"

  pr_con        "(#c)` == #c"
  pr_before     "($v)` == v$"
  pr_lift       "(F[x])` == F[x`]"
  pr_lift2      "(F[x,y])` == F[x`,y`]"
  pr_lift3      "(F[x,y,z])` == F[x`,y`,z`]"
  pr_all        "(RALL x. P(x))` == (RALL x. P(x)`)"
  pr_ex         "(REX x. P(x))` == (REX x. P(x)`)"

  unchanged_def "(unchanged v) [[s,t]] == (v t = v s)"
  square_def    "[A]_v == A .| unchanged v"
  angle_def     "<A>_v == A .& .~ unchanged v"

  enabled_def   "(Enabled A) s  ==  EX u. A[[s,u]]"
end
```

Figure 4: The Isabelle theory for the action level

```
actionI         (!!s t. ([[s,t]] |= A)) ==> A
actionD         A ==> ([[s,t]] |= A)
```

Basic action rules

```
idle_squareI    ([[s,t]] |= unchanged v) ==> ([[s,t]] |= [A]_v)
busy_squareI    ([[s,t]] |= A) ==> ([[s,t]] |= [A]_v)
```

Square actions

```
enabledI        A [[s,t]] ==> (Enabled A) s
enabledE        [| (Enabled A) s; !!u. A[[s,u]] ==> PROP R |] ==> PROP R
notEnabledD     ~ (Enabled G s) ==> ~ G [[s,t]]
enabled_mono    [| (Enabled F) s; F .-> G |] ==> (Enabled G) s
enabled_disj1   (Enabled F) s ==> (Enabled (F .| G)) s
enabled_disj2   (Enabled G) s ==> (Enabled (F .| G)) s
enabled_disjD   (Enabled (F .| G)) s ==> ((Enabled F) s) | ((Enabled G) s)
enabled_disj    (Enabled (F .| G)) s = ( (Enabled F) s | (Enabled G) s )
enabled_conj1   (Enabled (F .& G)) s ==> (Enabled F) s
enabled_conj2   (Enabled (F .& G)) s ==> (Enabled G) s
enabled_ex      (Enabled (REX x. F x)) s = (EX x. (Enabled (F x)) s)
base_enabled    [| base_var(v); !!u. v u = c s ==> A [[s,u]] |] ==> Enabled A s
```

Enabledness

Figure 5: Derived theorems of the action level

The priming function, externally represented as `'`, applies to arbitrary transition functions (including actions), and not only to state functions as in Lamport's TLA. Again, our pragmatic decision to keep the syntax simpler and more liberal is related to the absence of a separate logic of state functions in our representation. Even if nested primes occur in specifications or subgoal hypotheses, they can not be exploited in proofs. The (definitional) axioms `pr_con`, `pr_before`, `pr_lift` (and variants), `pr_all`, and `pr_ex` are used to push primes inward through complex expressions. (Since rigid quantification binds identifiers to values rather than state variables, the bound variable need not be changed.)

The shorthand notation for actions with square and angle brackets is as expected. More interestingly, theory `Action` defines the state predicate `Enabled A` via existential quantification over the second state of a state pair. These simple and transparent definitions of enabledness and of priming are the most visible benefits of our use of theory `Intensional` as a base for theory `Action`.

Figure 5 lists some derived theorems of the action level. Theory `Action` also defines unlifting functions analogous to those shown in figure 2 (whose names begin with `action_`), installs some basic action rewrites as default simplification rules, and sets up a "claset" and a "clasimpset" for Isabelle/HOL's automatic proof tactics. Automated tactics will be discussed in more detail in section 5.

4 The temporal level of TLA

We now describe our representation of the temporal level of TLA. In fact, we have really encoded a superset of TLA, called raw TLA in [10], which is not

```
TLA  =  Action + WF +

types
    behavior
    temporal = "behavior form"

arities
    behavior :: world

consts
  fst_st     :: "behavior => state"
  snd_st     :: "behavior => state"

  Init       :: "action => temporal"
  Box        :: "('w::world) form => temporal"      ("([](_))" [40] 40)
  Dmd        :: "('w::world) form => temporal"      ("(<>(_))" [40] 40)
  "~>"       :: "[action,action] => temporal"       (infixr 22)
  stable     :: "action => temporal"
  WF         :: "[action,'a stfun] => temporal"     ("(WF'(_')'_(_))" [0,60] 55)
  SF         :: "[action,'a stfun] => temporal"     ("(SF'(_')'_(_))" [0,60] 55)

  EEx        :: "('a stfun => temporal) => temporal"    (binder "EEX " 10)
  AAll       :: "('a stfun => temporal) => temporal"    (binder "AALL " 10)
```

Figure 6: The syntax of temporal logic

stuttering invariant. However, we believe that invariance under stuttering is mostly a concern when writing specifications, and that it is easier to perform proofs in raw TLA. One indication that supports this claim is the absence of satisfactory proof systems for TLA in the literature. (Lamport's proof system is complete only for specifications in so-called standard form.) Since TLA is a proper sublogic of raw TLA, any raw TLA proof about TLA formulas is also valid for TLA.

4.1 Embedding temporal logic in Isabelle

Figure 6 shows the first part of file `TLA.thy`, which defines the syntax of the temporal level of TLA. Observe that it is based on theories `Action` and `WF`, the standard formalization of well-founded relations in Isabelle/HOL. Although well-founded relations do not appear in the theory file, derived rules for proving liveness properties are based on well-founded relations (see section 4.3). The type `behavior` of class `world` represents state sequences, but it does not reveal the structure of its objects. The functions `fst_st` and `snd_st` intuitively select the first two states of a behavior in order to evaluate initial conditions. Formally, they are simply Skolem functions that name states. Function `Init` promotes actions to temporal formulas; such a type-conversion function is left implicit in textbook-style presentations of TLA. As with the priming function of theory `Action`, `Init` accepts any action, and not just state predicates, which is again

```
rules
  dmd_def    "(<>F) == .~[].~F"
  boxact_def "([](F::action)) == ([] Init F)"
  leadsto    "P ~> Q == [](Init(P) .-> <>Q)"
  stable_def "stable P == [](P .-> P`)"

  WF_def     "WF(A)_v == <>[] $(Enabled(<A>_v)) .-> []<><A>_v"
  SF_def     "SF(A)_v == []<> $(Enabled(<A>_v)) .-> []<><A>_v"

  Init_def   "(sigma |= Init(F)) == ([[fst_st sigma, snd_st sigma]] |= F)"

  normalT    "[](F .-> G) .-> ([]F .-> []G)"
  reflT      "[]F .-> F"
  transT     "[]F .-> [][]F"
  linT       "(<>F) .& (<>G) .-> (<>(F .& <>G)) .| (<>(G .& <>F))"
  discT      "[](F .-> <>(.~F .& <>F)) .-> (F .-> []<>F)"
  primeI     "[]P .-> Init(P`)"
  primeE     "[](Init(P) .-> []F) .-> Init(P`) .-> (F .-> []F)"
  indT       "[](Init(P) .& .~[]F .-> Init(P`) .& F) .-> Init(P) .-> []F"
  allT       "(RALL x. [](F(x))) .= ([](RALL x. F(x)))"

  necT       "F ==> []F"

  aall_def      "(AALL x. F(x)) == .~ (EEX x. .~ F(x))"
  eexI          "F x .-> (EEX x. F x)"
  historyI      "[| sigma |= Init(I); sigma |= []N;
                    (!!h s t. (h s = ha s t) & I [[s,t]] --> HI(h)[[s,t]]);
                    (!!h s t. (h t = hc s t (h s)) & N [[s,t]] --> HN(h) [[s,t]])
                 |] ==> sigma |= (EEX h. Init(HI(h)) .& []HN(h))"
  eexE          "[| sigma |= (EEX x. F x);
                    (!!x. [| base_var x; (sigma |= F x) |] ==> (G sigma)::bool)
                 |] ==> G sigma"
end
```

Figure 7: The rules of temporal logic

related to the absence of a logic of state predicates in our formalization. `Box` and `Dmd` (with external syntax `[]` and `<>`) are the "always" and "eventually" operators of temporal logic, `~>` represents the derived leads-to operator, `stable` is a useful abbreviation borrowed from UNITY, and `WF` and `SF` are the derived operators to express fairness properties. Note that `[]` and `<>` are defined over arbitrary intensional formulas, in particular, they apply to temporal formulas and actions so that one may conveniently write `[]A` instead of `[](Init A)` when `A` is an action.

Finally, theory `TLA` also provides binders `EEX` and `AALL` for flexible quantification over state variables.

Figure 7 shows the axioms and rules of temporal logic. The first group of definitional axioms needs no further explanation. The axioms `normalT` through `indT` and the rule `necT` form (a variant of) the proof system for propositional raw TLA whose completeness is proved in [12]. Axiom `allT` asserts that rigid universal quantification commutes with the box. These axioms are rarely used in actual verification tasks, where one relies instead on the (derived) rules described in section 4.3.

The remaining group of rules concerns flexible quantification. Axiom `eexI` is a formulation of the standard refinement-mapping rule; its formulation is analogous to the classical introduction rule for existential quantification. Rule `eexE` is an unlifted version of the analogous elimination rule; the bound variable may be treated as a "base" variable. Rule `historyI` allows to introduce an auxiliary (history) variable [1]; it is a translation of the textbook-style rule

$$\frac{\begin{array}{c} I \Rightarrow \exists h : HI \\ N \Rightarrow \exists h' : HN \end{array}}{I \wedge \Box N \Rightarrow \boldsymbol{\exists}\, h : HI \wedge \Box HN} \qquad (\textbf{history})$$

We cannot directly enter this rule into Isabelle because the variable h in the conclusion is a state variable whereas h and h' in the (action) premises are rigid variables; we have avoided to define a theory of substitution in temporal formulas. It should be a straightforward exercise to add a similar introduction rule for (non-stuttering) prophecy variables, but we have been reluctant to do so because we have not yet treated an example that uses prophecy variables.

4.2 Comparison with standard TLA

Our encoding differs from Lamport's presentation of TLA in at least two respects:

- Our formulation is typed, whereas standard TLA is untyped.
- We have represented (a superset of) raw TLA, which is not invariant under stuttering.

The first difference is not very essential in actual proofs: Isabelle's type system resembles that of functional programming languages like Haskell and is not a substitute for proving invariants. Nevertheless, types help to catch errors early on, and they do assist in proofs because they help to reduce the number of applicable auxiliary rules; also, facilities like the inductive datatypes of Isabelle/HOL come pre-equipped with specific rules (e.g., induction rules) that are helpful for verifications.

The second difference is more important, and it is mainly due to the absence of a satisfactory proof system for TLA. In particular, Lamport's rules [10] are designed for specifications in the "standard form" $I \wedge \Box[N]_v \wedge L$ where the predicate I describes the initial condition, N is the complete next-state relation, and L is a conjunction of fairness conditions. For example, they do not support specifications that contain several next-state relations (such formulas occur naturally when specifications are combined from subprocesses). Therefore, even Lamport sometimes advocates using raw TLA in proofs [11]. In fact, our embedding goes even beyond Lamport's raw TLA in that any action (and not just a state predicate) may be promoted to a temporal formula and in that nested occurrences of prime symbols are syntactically possible. As we have said

```
tempI          (!!sigma. sigma |= (F::temporal)) ==> F
tempD          F::temporal ==> (sigma |= F)
```

Basic introduction and elimination rules

```
Init_true      Init(#True) .= #True
Init_false     Init(#False) .= #False
Init_not       Init(.~P) .= (.~Init(P))
Init_and       Init(P .& Q) .= (Init(P) .& Init(Q))
Init_or        Init(P .| Q) .= (Init(P) .| Init(Q))
Init_imp       Init(P .-> Q) .= (Init(P) .-> Init(Q))
Init_iff       Init(P .= Q) .= (Init(P) .= Init(Q))
Init_all       Init(RALL x. P(x)) .= (RALL x. Init(P(x)))
Init_ex        Init(REX x. P(x)) .= (REX x. Init(P(x)))
```

Pushing Init through connectives (use Init_simps)

```
STL2           []?F .-> ?F                       STL2b         []P .-> Init P
ImplDmd        F .-> <>F                         InitDmd       Init(P) .-> <>P
STL3           ([][]F) .= ([]F)                  DmdDmd        (<><>F) .= (<>F)
STL4           (F .-> G)  ==> ([]F .-> []G)
DmdImpl        (F .-> G)  ==> (<>F .-> <>G)
STL5           ([]F .& []G) .= ([](F .& G))
DmdOr          (<>(F .| G)) .= (<>F .| <>G)
exT            (REX x. <>(F x)) .= (<>(REX x. F x))
STL4Edup       [| (sigma |= []A); (sigma |= []F); F .& []A .-> G |] ==> (sigma |= []G)
DmdImpl2       [| (sigma |= <>F); (sigma |= [](F .-> G)) |] ==> (sigma |= <>G)"
InfImpl        [| (sigma |= []<>F); (sigma |= []G); F .& G .-> H |] ==> (sigma |= []<>H)
BoxDmdT        []F .& <>G .-> <>([]F .& G)
STL6           <>[]F .& <>[]G .-> <>[](F .& G)
NotBox         (.~[]F) .= (<>.~F)                NotDmd        (.~<>F) .= ([].~F)
BoxDmdBox      ([]<>[]F) .= (<>[]F)              DmdBoxDmd     (<>[]<>F) .= ([]<>F)
DBImplBD       <>[](F::temporal) .-> []<>F       DBImplBDAct   <>[](A::action) .-> []<>A
BoxDmdDmdBox   [| (sigma |= []<>F); (sigma |= <>[]G) |] ==> (sigma |= []<>(F .& G))
```

Simple temporal logic

```
BoxPrime       []P .-> [](P .& P`)               DmdPrime      (<>P`) .-> (<>P)
TLA2           P .& P` .-> Q  ==>  []P .-> []Q
INV1           Init(P) .& stable(P) .-> []P
StableL        (P .& A .-> P`) ==> ([]A .-> stable(P))
StableBox      (sigma |= stable P) ==> (sigma |= [](Init P .-> []P))
DmdStable      [| (sigma |= stable P); (sigma |= <>P) |] ==> (sigma |= <>[]P)
unless         (sigma |= [](P .-> P` .| Q`)) ==> (sigma |= stable P .| <>Q`)
BoxRec         ([]$P) .= (Init($P) .& ([]P$))
DmdRec         (<>$P) .= (Init($P) .| (<>P$))
InfinitePrime  ([]<>$P) .= ([]<>P$)
```

Priming and recursive expansions

```
WF_alt         WF(A)_v .= (([]<>.~$(Enabled(<A>_v))) .| []<><A>_v)
SF_alt         SF(A)_v .= ((<>[].~$(Enabled(<A>_v))) .| []<><A>_v)
WF_Box         ([]WF(A)_v) .= WF(A)_v            SF_Box        ([]SF(A)_v) .= SF(A)_v
SFImplWF       (sigma |= SF(A)_v) ==> (sigma |= WF(A)_v)
```

Fairness

Figure 8: Derived TLA rules (part 1)

```
leadsto_init        [| (sigma |= Init P); (sigma |= P ~> Q) |] ==> (sigma |= <>Q)
leadsto_infinite    [| (sigma |= []<>P); (sigma |= P ~> Q) |] ==> (sigma |= []<>Q)
INV_leadsto         [| (sigma |= []I); (sigma |= (P .& I) ~> Q) |] ==> (sigma |= P ~> Q)
leadsto_classical   (sigma |= [](Init P .& [].~Q .-> <>Q)) ==> (sigma |= P ~> Q)
ImplLeadsto         (sigma |= [](P .-> Q)) ==> (sigma |= (P ~> Q))
EnsuresLeadsto      A .& P .-> Q` ==> []A .-> (P ~> Q)
EnsuresInfinite     [| (sigma |= []<>P); (sigma |= []A); A .& P .-> Q` |]
                    ==> (sigma |= []<>Q)
```

Leadsto

```
LatticeReflexivity        F ~> F
LatticeTransitivity       [| (sigma |= G ~> H); (sigma |= F ~> G) |] ==> (sigma |= F ~> H)
LatticeDisjunctionElim1   (sigma |= (F .| G) ~> H) ==> (sigma |= F ~> H)
LatticeDisjunctionElim2   (sigma |= (F .| G) ~> H) ==> (sigma |= G ~> H)
LatticeDisjunctionIntro   [| (sigma |= F ~> H); (sigma |= G ~> H) |]
                          ==> (sigma |= (F .| G) ~> H)
LatticeDiamond            [| (sigma |= B ~> D); (sigma |= A ~> (B .| C));
                             (sigma |= C ~> D) |]  ==>  (sigma |= A ~> D)
LatticeTriangle           [| (sigma |= B ~> D); (sigma |= A ~> (B .| D)) |]
                          ==> (sigma |= A ~> D)
```

Lattice rules

```
WF1   [| P .& N  .-> P` .| Q`;  P .& N .& <A>_v .-> Q`;
         P .& N .-> $(Enabled(<A>_v))
      |] ==> []N .& WF(A)_v .-> (P ~> Q)
SF1   [| P .& N  .-> P` .| Q`;  P .& N .& <A>_v .-> Q`;
         []P .& []N .& []F .-> <>$(Enabled(<A>_v))
      |] ==> []N .& SF(A)_v .& []F .-> (P ~> Q)
WF2   [| N .& <B>_f .-> <M>_g;  P .& P` .& <N .& A>_f .-> B;
         P .& $(Enabled(<M>_g)) .-> $(Enabled(<A>_f));
         [](N .& [.~B]_f) .& WF(A)_f .& []F .& <>[]($(Enabled(<M>_g))) .-> <>[]P
      |] ==> []N .& WF(A)_f .& []F .-> WF(M)_g
SF2   [| N .& <B>_f .-> <M>_g;  P .& P` .& <N .& A>_f .-> B;
         P .& $(Enabled(<M>_g)) .-> $(Enabled(<A>_f));
         [](N .& [.~B]_f) .& SF(A)_f .& []F .& []<>($(Enabled(<M>_g))) .-> <>[]P
      |] ==> []N .& SF(A)_f .& []F .-> SF(M)_g
```

Lamport's fairness rules

```
wf_dmd       [| (wf r); !!x. sigma |= [](F x .-> <>G .| <>(REX y. #((y,x):r) .& F y))
             |] ==> sigma |= [](F x .-> <>G)
wf_leadsto   [| (wf r); !!x. sigma |= P x ~> (Q .| (REX y. #((y,x):r) .& P y))
             |] ==> sigma |= P x ~> Q
wf_not_dmd_box_decrease  wf r  ==>  <>[][[v$, $v] .: #r]_v .-> <>[][#False]_v
wf_box_dmd_decrease      wf r ==> []<>([v$, $v] .: #r) .-> []<><.~([v$, $v] .: #r)>_v
```

Liveness by well-founded relations

```
aallI        (!!x. base_var x ==> sigma |= F x) ==> sigma |= (AALL x. F(x))
aallE        [| sigma |= (AALL x. F(x));  (!!sigma. sigma |= F(x) ==> P sigma) |]
             ==> (P sigma)::bool
eex_mono     [| sigma |= EEX x. F(x); !!x. F(x) .-> G(x) |] ==> sigma |= EEX x. G(x)
aall_mono    [| sigma |= AALL x. F(x); !!x. F(x) .-> G(x) |] ==> sigma |= AALL x. G(x)
```

Flexible quantification

Figure 9: Derived TLA rules (part 2)

before, this somewhat more liberal syntax helps to avoid a separate logic of state formulas and therefore makes the syntax lighter.

The reassuring point is that our proof system is correct not only for raw TLA, but also for TLA, even though non-TLA formulas may appear during the proof. One noticeable difference concerns the definition of quantification of flexible variables: the TLA definition explicitly ensures that quantified formulas are invariant under stuttering (see [10] for details), whereas that definition would not make sense for raw TLA (because a formula F would no longer be equivalent to $\boldsymbol{\exists}\, x : F$ when x does not appear in F). In particular, an introduction rule for "stuttering variables" that introduce additional stuttering would be unsound for raw TLA, implying that certain TLA proofs cannot be performed in our system. Therefore it remains interesting to look for sound and complete proof systems for TLA (or perhaps a stuttering-invariant superset of TLA).

4.3 Derived TLA theorems

Theory `TLA` contains many derived theorems that are used in actual proofs. Like theory `Action`, it provides specific versions of unlifting rules like those shown in figure 2, whose names begin with `temp_`, and it sets up a "claset" and a "clasimpset" for the automated provers. Figures 8 and 9 show derived TLA theorems. Some of these theorems are directly presented in their unlifted forms, which are generally more useful in verification. For most of the others, the theory also provides unlifted variants; please refer to the HTML documentation for a complete list of derived rules. It is one of the advantages of basing the encoding on a complete proof system that we can actually derive these theorems. As far as we know, most other encodings have stated many of these theorems as axioms. Observe also that we benefit from standard Isabelle/HOL tools such as the pre-existing formulation of well-founded relations.

5 Additional tactics for verification

We have carried out a small number of example verifications in our system and have programmed a number of tactics to do so. However, these are probably not stable; much more experience is necessary to write useful tactics.

The principal automated reasoning tools offered by Isabelle are its rewriting-based simplifier, the classical prover that automates reasoning in mathematical logic, and, more recently, the integration of these two tools. Our representation builds on the instantiation of these tools in Isabelle/HOL.

Theory `Intensional` adds its unlifting rules and a number of (mainly propositional) simplifications at the intensional level to the default set of simplifications. It also sets up the simplifier such that it accepts "intensional" goals and immediately unlifts them (using `int_unlift`).

Theory `Action` adds a number of simplifications of its own. More importantly, it defines a new alternative substitution tactic `action_hyp_subst_tac` and sets up both a classical set (`action_cs`) and a "clasimpset" that use it. The effect of this tactic is to use equations like `x s = 0` in the antecedent of a subgoal to substitute `0` for `x s` throughout the goal, provided `s` is of type `state`. (The default `hyp_subst_tac` of Isabelle/HOL only substitutes equations of the form `x = 0`, not applications.) It also adds the rules `actionI` and `intI` as safe introduction rules, so that intensional or action-level goals are automatically unlifted, and `actionD` and `intD` as unsafe elimination rules, so that lifted hypotheses can be used in proofs.

Theory `Action` also defines a simplification-based tactic `action_simp_tac` that applies (safe) introduction and elimination rules when the simplifier is stuck. Although this tactic is much more simplistic than Isabelle's new `auto_tac`, it can be much faster in situations where blind application of simple rules is successful.

Finally, `enabled_tac` is designed to prove simple enabledness conditions. Its first argument should specify a list of base variables.

Theory `TLA`, which defines the temporal level, builds on the tactics defined in theory `Action` by adding a number of temporal simplification rules and defining a classical set `temp_cs` and a "clasimpset" `temp_css`.

The tactic `merge_box_tac` collects all boxed formulas that appear in the hypotheses of a subgoal in the scope of a single box; this is often useful before applying rules like `STL4`. In some cases (when the hypotheses contain both boxed temporal formulas and boxed action formulas), it may be too polymorphic; use its variants `merge_temp_box_tac` or `merge_act_box_tac`, as appropriate.

Invariants have a particularly stylized proof, which is supported by `inv_tac` and `auto_inv_tac`. The latter tactic attempts to solve the subgoals by simplification. In simple cases (such as the "increment" example [10]) it can prove an invariant without further assistance.

At the moment, there are no high-level tactics for liveness proofs, because these proofs do not usually follow a clear-cut pattern. It would probably be more appropriate to have a higher-level language of proof hints, such as the verification diagrams offered by the STeP prover [3].

6 A complete example

We conclude the description of our TLA embedding with a simple example that shows the overall style of reasoning in our system.

Theory `Demo`, shown in figure 10, imports theories `TLA` and `Even`, which defines even numbers by

```
even_def "even n   ==   EX m. n = m+m"
```

and proves lemmas such as

```
val even_suc = "even ?n ==> even (Suc (Suc ?n))" : thm
```

We wish to prove that `x` and `y` are always even in runs of program `Phi`:

```
goal Demo.thy "Phi .-> [](even[$x] .& even[$y])";
   Level 0
    1. Phi .-> [](even[$x] .& even[$y])
```

```
Demo  =  TLA + Even +

consts
  x,y                   :: "nat stfun"
  PhiInit               :: "action"
  IncX,IncY             :: "action"
  Phi                   :: "temporal"

rules
  PI_def     "PhiInit  == ($x .= # 0) .& ($y .= # 0)"
  IncX_def   "IncX     == x$ .= $x .+ # 2"
  IncY_def   "IncY     == y$ .= $y .+ # 2"
  Phi_def    "Phi      == Init(PhiInit) .& [][IncX]_x .& [][IncY]_y"
  Phi_base   "base_var <x,y>"
end
```

Figure 10: Theory `Demo`

Isabelle's auto-tactic unlifts the goal and performs some initial simplifications:

```
auto();
   Level 1
   1. !!sigma. Phi sigma ==> sigma |= [](even[$x] .& even[$y])
```

We apply the unlifted version of rule `INV1`:

```
by (rtac INV1I 1);
   Level 2
    1. !!sigma. Phi sigma ==> sigma |= Init (even[$x] .& even[$y])
    2. !!sigma. Phi sigma ==> sigma |= stable (even[$x] .& even[$y])
```

which leaves us with proving the invariant in the initial state and proving that it remains stable across the specification's next-state relation. The simplification-based `action_simp_tac` takes care of the first subgoal:

```
by (action_simp_tac (!simpset addsimps [Phi_def,PI_def,Init_def,even_0]) [] [] 1);
   Level 3
    1. !!sigma. Phi sigma ==> sigma |= stable (even[$x] .& even[$y])
```

It is time to expand the definition of `Phi`

```
by (auto_tac (temp_css addsimps2 [Phi_def]));
   Level 4
    1. !!sigma.
          [| sigma |= Init PhiInit; sigma |= [][IncX]_x;
             sigma |= [][IncY]_y |]
          ==> sigma |= stable (even[$x] .& even[$y])
```

and combine the two next-state relations:

```
by (merge_box_tac 1);
   Level 5
    1. !!sigma.
          [| sigma |= Init PhiInit; sigma |= []([IncX]_x .& [IncY]_y) |]
          ==> sigma |= stable (even[$x] .& even[$y])
```

(Observe that `[]([IncX]_x .& [IncY]_y)` is not a TLA formula according to Lamport's definition.) Now, elim-resolution with `Stable` leaves us with proving another action formula, which `action_simp_tac` can handle:

```
by (etac Stable 1);
   Level 7
   Phi .-> [](even[$x] .& even[$y])
    1. !!sigma.
          sigma |= Init PhiInit
          ==> (even[$x] .& even[$y]) .& [IncX]_x .& [IncY]_y .->
              (even[$x] .& even[$y])`

by (action_simp_tac (!simpset addsimps [square_def,IncX_def,IncY_def])
                    [even_Suc] [] 1);
   Level 7
   No subgoals!
```

The invariant tactic `auto_inv_tac` is designed to handle such simple invariants automatically.

7 Conclusion

The work reported in this paper represents an ongoing experiment with the hope to use Isabelle as a tool to verify TLA specifications of at least academic size. Our experience so far seems to indicate that this is indeed feasible. Several other embeddings of TLA in interactive theorem provers have been described in the literature, and we now try to compare some of these with our work.

Probably most widely used has been TLP [5], which consists of a custom-built SML parser that interprets both formulas and proofs, an Emacs front-end and designed to handle various provers as back-ends (it presently supports the Larch prover and a BDD-based procedure to check propositional tautologies of linear temporal logic). Although this design is very flexible in principle, it is not clear how hard it really is to add a new back-end. Besides, TLP has a few drawbacks. For example, enabledness predicates cannot be computed but have to be supplied by the user.

Kalvala [7] describes another embedding of TLA in Isabelle. Although she defines states and behaviors in Isabelle/ZF, Lamport's rules are introduced axiomatically. State variables are restricted to be natural numbers. From a user's point of view, it appears that some subtleties of the semantic representation penetrate to the user level, as far as one can judge from the examples in her paper. Another semantic embedding of TLA in the HOL prover described by von Wright and Långbacka [8, 16] appears to be more oriented towards meta-theoretic proofs about TLA than towards system verification.

Finally, Busch [4] describes a syntactic embedding of TLA in the theorem prover Lambda. Unlike TLP, it tries to separate modal contexts using conversion functions that "clean up" the proof state. This approach has two potential problems: First, the correctness of the system depends not only on the explicitly stated axioms and rules, but also on extra-logical (procedural) features that may be hard to verify. Second, having to apply the conversion functions at every step could degrade the performance of the system.

It appears that our system is the first one to provide quantification of flexible variables.

We have found it very useful to define a generic theory of "intensional" logics, trivial as it may seem, that captures the concept of reasoning about possible worlds without having to reveal more about the structure of worlds than necessary. This separation keeps the design modular and supports a spectrum of representations rather than just the extreme cases of deep or shallow embeddings. Basin et al [2] have developed a superficially similar theory of modal logics in Isabelle, based on Gabbay's framework of labelled deductive systems. In contrast to our embedding, they explicitly represent the accessibility relation between worlds and axiomatize that relation using Horn clauses. We cannot use their theory for TLA because linearity and discreteness are not definable by Horn clauses. More fundamentally, we do not store the relationships between different worlds, which means that all relevant formulas have to be transported to a new world as soon as it is introduced. This makes reasoning in our embedding resemble more Hilbert-style than natural deduction-style calculi, and is probably not very adequate for complex temporal reasoning. Fortunately, one design philosophy of TLA is to minimize the amount of temporal reasoning necessary, which should alleviate the problem as far as system verification is concerned.

The emphasis in our approach is on a standardized style of proofs that delegates most of the work to the level of the base logic, all of whose rules remain valid. For example, we have benefited from predefined theories like arithmetics, existing tactics, and facilities such as Isabelle/HOL's package of inductive data types.

Of course, much work remains to be done. From a user's point of view, one of the main drawbacks of tactic-based provers like Isabelle is the lack of a higher-level proof language instead of procedural tactics. The hierarchical proof style described in [9] and supported by TLP looks like an interesting suggestion to explore. The power of our prover could be greatly enhanced by an integration of automatic decision procedures and model checkers. A standard interface to link to external "oracles" has recently been added to Isabelle. We have some encouraging preliminary results with a theory of abstractions that would compute a finite-state, abstract version of a specification, which could then be checked automatically. This approach may be most promising for proving liveness properties, which are very tedious to verify "by hand".

Acknowledgements. Larry Paulson and Markus Wenzel suggested improvements to the Isabelle theories and this documentation.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 81(2):253–284, May 1991.

[2] David Basin, Sean Matthews, and Luca Vigano. Labelled propositional modal logics: Theory and practice. Technical Report MPI-I-96-2-002, Max-Planck-Institut für Informatik, Saarbrücken, 1996. To appear in the Journal of Logic and Computation.

[3] Nikolaj Bjørner, Anca Browne, Edward Chang, Michael Colón, Arjun Kapur, Zohar Manna, Henny B. Sipma, and Tomás E. Uribe. STeP: deductive-algorithmic verification of reactive and real-time systems. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification: 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418. Springer-Verlag, 1996.

[4] Holger Busch. A practical method for reasoning about distributed systems in a theorem prover. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Workshop on Higher-Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 106–121, Berlin, 1995. Springer-Verlag.

[5] Urban Engberg. *Reasoning in the Temporal Logic of Actions.* PhD thesis, Aarhus University, 1994. See also: Preliminary TLP manual. Available at http://www.daimi.aau.dk/~urban/tlp/tlp.html.

[6] Stephen J. Garland and John V. Guttag. Lp: the larch prover, version 3.1. Technical report, MIT Laboratory for Computer Science, January 1995. Available at URL http://larch-www.lcs.mit.edu:8001/larch/LP/overview.html.

[7] Sara Kalvala. A formulation of TLA in Isabelle. Available at http://www.dcs.warwick.ac.uk/~sk/papers/tla.dvi, March 1995.

[8] Thomas Långbacka. A HOL formalisation of the Temporal Logic of Actions. In Thomas E Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 332–345, Berlin, 1994. Springer-Verlag.

[9] Leslie Lamport. How to write a proof. Research Report 94, Digital Equipment Corporation, Systems Research Center, February 1993.

[10] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[11] Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, September 1994.

[12] Stephan Merz. On the completeness of propositional raw TLA. Available at URL http://www4.informatik.tu-muenchen.de/~merz/isabelle/TLA/doc/ptla.ps, 1997.

[13] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[14] Lawrence C. Paulson. *Logic and Computation: Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.

[15] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, Heidelberg, 1994.

[16] J. von Wright. Mechanizing the Temporal Logic of Actions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *1991 International Workshop on the HOL Theorem Proving System and its Applications.* IEEE Computer Society Press, 1992.