The Semantics of TLA
on the PVS Theorem Prover

M. Wahab
Department of Computer Science
University of Warwick

October 4, 1996

Abstract

An implementation of Lamport's Temporal Logic of Actions (TLA) on a higher order logic theorem prover is described. TLA is a temporal logic, for which a syntax and semantics are defined, based on an action logic which is represented by higher order functions. The temporal logic includes quantifiers for variables with constant values and for variables whose values change over time. The semantics of the latter depend on an auxiliary function which cannot be defined by primitive recursion and an alternative is given based on the Hilbert ϵ operator.

1 Introduction

The Temporal Logic of Actions (Lamport, 1994) is a system for reasoning about programs by considering the changes made to program variables during an execution. Actions are boolean expressions relating the values of the variables before and after some event, typically the execution of a command, of the program. Propositional operators are defined on actions giving the base of the temporal logic (the modal system S4.3.1, see Abadi, 1993; Rescher & Urquahart, 1971). This adds the unary operator always, $\Box$, such that if $\Box F$ is true at some point during the program execution then F is true at that and every subsequent point. Two existential quantifiers are defined in TLA: the first, over logical variables, has its usual meaning as an abstraction operator which hides a constant value; the second, over variables of the program, hides the values assigned to a variable during the program's execution. The quantifier free logic is known as Simple TLA and includes a number of nonstandard operators for reasoning about actions. With a single exception these are derivable from the temporal and propositional operators.

There have been a number of implementations of Simple TLA on theorem provers: TLP (Engberg et al., 1992) uses the first order logic of the Larch prover (Garland & Guttag, 1991) to verify the action logic and a separate verifier for the temporal logic; other implementations used the higher order logic theorem provers HOL (von Wright & Långbacka, 1992) and Isabelle/ZF (Kalvala, 1995). Proof rules for TLA have been given by Busch (1995) as axioms for the Lambda theorem prover and for Simple TLA by Engberg (1995) for the TLP prover.

A semantics for TLA requires a quantifier over program variables and Lamport (1994) defines this using an auxiliary function which is not primitive recursive. Långbacka (1994) gives an alternative definition of the quantifier, suitable for use in data refinement, but suggests that this may not be as general as that of Lamport. Using a method similar to temporal abstraction (Melham, 1993), an uncomputable function can be defined which is equivalent to, and a specification, of the auxiliary function. The semantics of the TLA quantifier can then be defined in higher order logic using this specification.

The implementation is given for PVS (Owre, Rushby & Shankar, 1993), an interactive theorem prover based on typed higher order logic with a specification language supporting recursive function and type definitions. **Section 2** briefly describes those parts of the specification language used in the implementation. The syntax and semantics of TLA are summarised in **section 3** and **section 4** describes its implementation on PVS. The implementation uses temporal abstraction to specify the existential quantifier and a proof of the equivalence of the auxiliary function and its specification is given in the **appendix**. **Section 5** gives an example of how a TLA formula may be presented in the implementation and is followed by the **conclusions**.

Notation. Operators: The boolean true, conjunction (and), disjunction (or), negation (not), implication (implies) and equivalence (iff) operators are, in general and when referring to the PVS logic, given as *true*, $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and $\Leftrightarrow$ respectively. The symbols $\forall$ and $\exists$ denote the universal and existential quantifiers and appear as e.g. $\forall(x : T) : P(x)$, where T is the type of x, or as $\forall x : P(x)$. The form $P[v/x]$ denotes the textual substitution of v for free occurrences of x in P. Functions are given in lower case or in bold type and predicates end with a question mark; *eg* and **eg** are functions and *eg?* is a predicate. Defined types are usually written with a leading capital although other forms are used when there is no ambiguity.

2 The PVS specification language

A PVS specification is based on modules, called *theories*, in which types, functions and variables are specified and logical formulae given as axioms or theorems. Functions and types can be passed to modules as arguments and can also be referred to by other modules.

A type is either declared without definition or is defined by enumeration, as an abstract data type or as a subtype by predicates on the values of an existing type. Types provided by PVS include the natural numbers, booleans and infinite sequences of a given type. Tuples have type $(A_1 \times \ldots \times A_n)$ where each A_i is a type which may be distinct from A_j ($j \neq i$). A term of the form $c : T$ declares the constant c with type T.

The definition of an abstract data type (ADT), which may be parameterised, generates a theory containing the definitions and declarations needed for its use. The definition of an ADT has the form

$$
\begin{array}{l}
\textit{name}[p_1, \ldots, p_n] : \texttt{datatype} \\
\texttt{begin} \\
\textit{cons}_1(\textit{acc}_1, \ldots, \textit{acc}_k) : \textit{rec}_1 \\
\qquad\qquad \vdots \\
\textit{cons}_m(\textit{acc}_1, \ldots, \textit{acc}_j) : \textit{rec}_m \\
\texttt{end}\ \textit{name}
\end{array}
$$

The ADT identifier is *name*, each p_j is a parameter to the type, each $\textit{cons}_i$ is a constructor and each $\textit{acc}_l$ is an accessor function returning a value of some type. Each $\textit{rec}_i$ is a predicate on instances of the ADT acting as a recogniser for the associated constructor, $\textit{cons}_i$, such that $\textit{rec}_i(x)$ for x of type *name* is true iff x is constructed with $\textit{cons}_i$.

Functions have types $(A \rightarrow B)$ and are defined as lambda expressions (and sugared equivalents). Functions and some operators may be overloaded, e.g. the functions $f : (\textit{nat} \rightarrow \textit{boolean})$ and $f : ((\textit{nat} \times \textit{nat}) \rightarrow \textit{nat})$ are distinct. Predicates are functions returning booleans and sets of a type are treated as predicates on elements of that type. Recursive functions must have a 'measure' function to show that the function terminates. A limited form of polymorphism is supported: if type T is an argument to a theory module, functions can be defined with types dependent on T, which will be instantiated when the functions are used.

A conditional is provided by an `if-then-else-endif` expression for which both branches must be given. An expression for pattern matching on an instance of an ADT is available and is equivalent to a nested conditional using the recognisers of the ADT.

Logical formulae can be presented as axioms, which are assumed to be true, or theorems, which must be proved. A logical formula is any expression which returns a boolean value. The usual operators are available and include equivalence, equality and the universal and existential quantifiers. Formulae expressing boolean equivalence can be used by the prover as rewrite rules.

3 TLA

The domain of TLA is the set of states which are modeled as functions from a set of identifiers to a set of values. A program variable, whose value may change, is named with an identifier from the set of identifiers or with an identifier from the set combined with a prime (if x is a variable then so is x'). The values of the program variables are given by some state. Rigid (logical) variables have constant values and are independent of the states.

Expressions using program variables are called state functions and their value is dependent on the values of the program variables. A boolean expression in which all program variables are unprimed is a predicate. A primed predicate or function is one in which all program variables are primed: if P is a predicate, P' is a primed predicate. A boolean expression which uses both unprimed and primed variables is an action and an expression containing a temporal operator is a temporal formula.

The semantics of the action logic assumes the set of identifiers *Vars* and the set of values *Vals*.

Definition 1 *States and Behaviours*

A state is a total function from *Vars* to *Vals*, $St \stackrel{\text{def}}{=} (\mathit{Vars} \rightarrow \mathit{Vals})$.

A state function is a function which takes a state as an argument and returns an element of some type T. The type of state functions is $(\mathit{St} \rightarrow T)$

A sequence of a type T is a total function from the naturals to T, $\mathit{Seq}[T] \stackrel{\text{def}}{=} (\mathit{nat} \rightarrow T)$ and a behaviour is a sequence of states, $\mathit{Behaviour} \stackrel{\text{def}}{=} \mathit{Seq}[\mathit{St}]$

If σ is a behaviour and n is a natural then $\sigma(n)$ is the nth state in the behaviour and σ_n is the behaviour obtained by dropping the first $n-1$ states in σ.

$$\sigma_n \stackrel{\text{def}}{=} (\lambda(m : \mathit{nat}) : \sigma(n+m))$$

The concatenation of a state s to the beginning of a behaviour σ is written $s \circ \sigma$ and, for a natural number n, satisfies

$$(s \circ \sigma)(n) = \begin{cases} s & \text{if } n = 0 \\ \sigma(n-1) & \text{if } n > 0 \end{cases}$$

□

A TLA formula is made up of actions or TLA formulae combined with a logical operators

Definition 2 *Formulae of TLA*

An action is a boolean expression, possibly containing rigid variables, unprimed program variables or primed program variables. If A is an action then so is $\mathrm{Enabled}(A)$.

$$
\begin{array}{llcl}
\text{(Leads to)} & F \leadsto G & \stackrel{\text{def}}{=} & \Box(F \Rightarrow \Diamond G) \\
\text{(Unchanged)} & \text{Unchanged}(f) & \stackrel{\text{def}}{=} & f' = f \\
\text{(Stuttering)} & [A]_f & \stackrel{\text{def}}{=} & A \vee \text{Unchanged}(f) \\
 & \langle A \rangle_f & \stackrel{\text{def}}{=} & A \wedge \neg\text{Unchanged}(f) \\
\text{(Fairness)} & \text{WF}_f(A) & \stackrel{\text{def}}{=} & \Box\Diamond \langle A \rangle_f \vee \Box\Diamond\neg\text{Enabled}\,\langle A \rangle_f \\
 & \text{SF}_f(A) & \stackrel{\text{def}}{=} & \Box\Diamond \langle A \rangle_f \vee \Diamond\Box\neg\text{Enabled}\,\langle A \rangle_f
\end{array}
$$

where F and G are temporal formulae, A is an action and f is a state function.

Figure 1: Derived operators of TLA

The syntax of a TLA formula is

$$T \ ::= \text{an action} \mid \neg T \mid T \wedge T \mid \Box T \mid \exists x : T \mid \exists v : T$$

where x is a rigid variable and v is an unprimed program variable.

A syntactically correct temporal formula is called a well-formed formula (wff) of TLA. □

Other logical operators ($\vee$, $\Rightarrow$, etc) are derived from the negation and conjunction operators in the usual way. The dual of the always ($\Box$) operator is the eventually operator ($\Diamond$) and $\Diamond F = \neg\Box\neg F$. The definitions of the derived TLA operators are given in Figure 1.

The semantics of TLA are given by an interpretation function mapping well-formed formulae and behaviours to booleans.

Definition 3 *Semantics of TLA*

The interpretation function, $\mathcal{I}$, for action A and states s and t is defined as

$$
\begin{array}{rl}
\mathcal{I}(A, s, t) & \stackrel{\text{def}}{=} \forall(v : \mathit{Vars}) : A[s(v)/v][t(v)/v'] \ (A \text{ is a boolean expression}) \\
\mathcal{I}(\text{Enabled(A)}, s, t) & \stackrel{\text{def}}{=} \exists(u : \mathit{St}) : \mathcal{I}(A, s, u)
\end{array}
$$

For TLA formulae F and G and behaviour σ, $\mathcal{I}$ is defined

$$
\begin{array}{rl}
\mathcal{I}(F, \sigma) & \stackrel{\text{def}}{=} \mathcal{I}(F, \sigma(0), \sigma(1)) \ (F \text{ is an action}) \\
\mathcal{I}(\neg F, \sigma) & \stackrel{\text{def}}{=} \neg\mathcal{I}(F, \sigma) \\
\mathcal{I}(F \wedge G, \sigma) & \stackrel{\text{def}}{=} \mathcal{I}(F, \sigma) \wedge \mathcal{I}(G, \sigma) \\
\mathcal{I}(\Box F, \sigma) & \stackrel{\text{def}}{=} \forall(n : \mathit{nat}) : \mathcal{I}(F, \sigma_n) \\
\mathcal{I}(\exists x : F, \sigma) & \stackrel{\text{def}}{=} \exists(c \in \mathit{Vals}) : \mathcal{I}(F[c/x], \sigma)
\end{array}
$$

□

The interpretation function defines the semantics of predicates and primed predicates as instances of actions. For a predicate P and states s and t, the interpretation function gives $\mathcal{I}(P, s, t) = \forall(v : \mathit{Vars}) : P[s(v)/v]$ and $\mathcal{I}(P', s, t) = \forall(v : \mathit{Vars}) : P[t(v)/v']$.

The existential quantifier over program variables creates an abstraction of a temporal formula in which the value of the quantified variable is hidden. If, for some program variable v, the formula $F(v)$

is true in a behaviour σ then $\exists v : F(v)$ is true in any behaviour β which differs from σ only in the values of v, since there is a sequence of values which may be assigned to v in each state of β to make $F(v)$ true in β. The temporal logic is based on actions and $F(v)$ may depend on the values of variables in consecutive states. Since β may differ from σ only in the value of v, this imposes the constraint that for every natural i, $\beta(i) \neq \beta(i+1)$ iff $\sigma(i) \neq \sigma(i+1)$. This is too strong: when v is the only variable whose value changes, the interpretation of $\exists v : F(v)$ in β will depend on changes to the value of v in σ and therefore on the value of v. To remove this dependence, only the states of β and σ which are not equal to their successors are considered.

Definition 4 *Existential quantifier over program variables*

An equality, $=_v$, between a pair of behaviours, takes a program variable v and compares all states in the behaviours up to v.

$$\sigma =_v \beta \stackrel{\text{def}}{=} \forall (n : \mathit{nat}) : \forall (x \in \mathit{Vars}) : x \neq v \Rightarrow \sigma(n)(x) = \beta(n)(x)$$

A reduction function takes a behaviour and removes all states which are equal to their successors.

$$\natural\sigma \stackrel{\text{def}}{=} \begin{cases} \sigma & \text{if } \forall (n : \mathit{nat}) : \sigma(n) = \sigma(0) \\ \natural\sigma_1 & \text{if } \sigma(0) = \sigma(1) \\ \sigma(0) \circ \natural\sigma_1 & \text{if } \sigma(0) \neq \sigma(1) \end{cases}$$

For an unprimed program variable v, the interpretation of the existential quantifier is defined as

$$\mathcal{I}(\exists v : F, \sigma) \stackrel{\text{def}}{=} \exists (\beta, \tau : \mathit{Behaviour}) : (\natural\sigma = \natural\beta) \wedge (\beta =_v \tau) \wedge \mathcal{I}(F, \tau)$$

□

4 Implementation of TLA in PVS

Given the sets *Vars*, of variables, and *Vals*, of values, the types used in the implementation are as defined for the TLA semantics. TLA formulae are represented by mapping actions to predicates in the PVS logic and by defining the type of well-formed temporal formulae. The semantics of TLA formulae are given by a function from the well-formed formulae to the booleans.

State functions are represented as PVS functions from states to some type T. TLA defines operators on state functions and the representation of these operators must be defined on functions returning any type. A theory module is declared with type T as a parameter and a type *tla_Fn* contains the functions from the states to T. The representation of the TLA operators are then defined on the *tla_Fn* type: e.g. the representation of the *unchanged* operator takes an argument of type *tla_Fn*$[T]$ and can be specialised to, among others, $T = (\mathit{nat} \times \mathit{nat})$ and $T = \mathit{boolean}$.

Actions and predicates

Actions are represented as predicates on state pairs.

Definition 5 *Type of actions*

Actions is the type containing functions from pairs of states to the booleans,

$$\mathit{Actions} \stackrel{\text{def}}{=} ((\mathit{St} \times \mathit{St}) \rightarrow \mathit{boolean})$$

□

Predicates are state functions of type $(St \rightarrow boolean)$ and may be promoted to elements of the type *Actions*. The promotion of a predicate P to an action is **unprimed**$(P) = (\lambda(s,t) : P(s))$ and the promotion of a primed predicate is **primed**$(P) = (\lambda(s,t) : P(t))$, where s and t are states.

Temporal Logic

The syntax of the temporal logic is given by an abstract data type, *wff*, containing the temporal formulae which are either actions or actions composed with a conjunction, negation or always ($\Box$) operator. An interpretation function maps elements of this type to the boolean values, defining the semantics.

Because of restrictions on the definition of abstract data types, the existential quantifiers cannot be defined directly but are described as predicates on behaviours. The syntax and semantics of the quantifiers are then defined by the parameters and definitions of these predicates.

Definition 6 *Type of temporal formulae*

The abstract data type *wff* has the base constructors **action**, on actions, and **exf**, on predicates over behaviours. It has the recursive constructors **not**, **and** and $\Box$ representing temporal formulae with negation, conjunction and always.

$$
\begin{array}{lll}
\textit{wff} : \texttt{datatype} & & \\
\texttt{begin} & & \\
\textbf{action} & (\textbf{ac} : \textit{Actions}) & : \textbf{action?} \\
\textbf{not} & (\textbf{neg} : \textit{wff}) & : \textbf{not?} \\
\textbf{and} & (\textbf{lcj} : \textit{wff}, \textbf{rcj} : \textit{wff}) & : \textbf{and?} \\
\textbf{always} & (\textbf{alw} : \textit{wff}) & : \textbf{always?} \\
\textbf{exf} & (\textbf{exw} : (\textit{Behaviour} \rightarrow \textit{boolean})) & : \textbf{exf?} \\
\texttt{end}\ \textit{wff} & &
\end{array}
$$

A temporal formulae has the type *tla_Formula*, and its subtype, *act_Formula*, contains only formulae representing actions.

$$
\begin{array}{l}
\textit{tla\_Formula} \stackrel{\text{def}}{=} \textit{wff} \\
\textit{act\_Formula} \stackrel{\text{def}}{=} \{F : \textit{tla\_Formula} \mid \textbf{action?}(F)\}
\end{array}
$$

$\Box$

Operators are written using infix notation, e.g. **and(A, B)** is given as A **and** B; the formula **always**(F) is written as $\Box F$.

The semantics of the temporal formulae are defined by the interpretation function, **trans**, mapping an element of type *tla_Formula* and a behaviour to the boolean values.

Definition 7 *Interpretation of temporal formulae*

The function **trans** has the type **trans** : $((\textit{tla\_Formula} \times \textit{Behaviour}) \rightarrow \textit{boolean})$ and is recursively defined as

$$
\begin{array}{rl}
\textbf{trans}(\textbf{action}(A), \sigma) & \stackrel{\text{def}}{=} A(\sigma(0), \sigma(1)) \\
\textbf{trans}(\textbf{not}(F), \sigma) & \stackrel{\text{def}}{=} \neg\textbf{trans}(F) \\
\textbf{trans}(\textbf{and}(F, G), \sigma) & \stackrel{\text{def}}{=} \textbf{trans}(F, \sigma) \wedge \textbf{trans}(G, \sigma) \\
\textbf{trans}(\Box F), \sigma) & \stackrel{\text{def}}{=} \forall (n : \textit{nat}) : \textbf{trans}(F, \sigma_n) \\
\textbf{trans}(\textbf{exf}(\mathcal{F}), \sigma) & \stackrel{\text{def}}{=} \mathcal{F}(\sigma)
\end{array}
$$

$\Box$

$$
\begin{array}{llllll}
\text{(Enabled)} & \textbf{enabled}(A) & : \textit{act\_Formula} & \stackrel{\text{def}}{=} & \textbf{action}(\textbf{unprimed}(\lambda s : (\exists t : A(s,t)))) \\
\text{(Unchanged)} & \textbf{unchanged}(f) & : \textit{act\_Formula} & \stackrel{\text{def}}{=} & \textbf{action}(\lambda(s,t) : f(t) = f(s)) \\
\text{(Eventually)} & \Diamond F & : \textit{tla\_Formula} & \stackrel{\text{def}}{=} & \textbf{not}\ \Box\ \textbf{not}\ F \\
\text{(Leads to)} & \leadsto (F, G) & : \textit{tla\_Formula} & \stackrel{\text{def}}{=} & \Box(F\ \textbf{implies}\ \Diamond G) \\
\text{(Stuttering)} & \textbf{square}(A, f) & : \textit{tla\_Formula} & \stackrel{\text{def}}{=} & (A\ \&\ \textbf{action}(\lambda(s,t) : f(t) = f(s))) \\
 & \textbf{angle}(A, f) & : \textit{tla\_Formula} & \stackrel{\text{def}}{=} & (A \mid -\textbf{action}(\lambda(s,t) : f(t) = f(s))) \\
\text{(Fairness)} & \textbf{WF}(A, f) & : \textit{tla\_Formula} & \stackrel{\text{def}}{=} & \Box\Diamond(\textbf{angle}(A, f)) \\
 & & & & \quad \textbf{or}\ \Box\Diamond\ \textbf{not}\ (\textbf{enabled}(\textbf{angle}(A, f))) \\
 & \textbf{SF}(A, f) & : \textit{tla\_Formula} & \stackrel{\text{def}}{=} & \Box\Diamond(\textbf{angle}(A, f)) \\
 & & & & \quad \textbf{or}\ \Diamond\Box\ \textbf{not}\ (\textbf{enabled}(\textbf{angle}(A, f)))
\end{array}
$$

where f : *tla_Fn*, A : *act_Formula* and F, G : *tla_Formula.*

Figure 2: TLA derived operators

The boolean *true* is defined as an instance of the actions subtype, *act_Formula.* Negation and conjunction operators are also defined for the *act_Formula* subtype and are named to avoid ambiguity with the operators of the temporal formulae.

Definition 8 *Action operators*

The boolean **true** is the action which returns *true* for all states. The negation ($-$) and conjunction ($\&$) operators on action formulae create new instances of the *act_Formula* subtype.

$$
\begin{array}{rl}
\textbf{true} & : \textit{act\_Formula} \\
 & \stackrel{\text{def}}{=} \textbf{action}(\lambda(s, t : \textit{St}) : \textit{true}) \\
-(A) & : (\textit{act\_Formula} \rightarrow \textit{act\_Formula}) \\
 & \stackrel{\text{def}}{=} \textbf{action}(\lambda(s, t : \textit{St}) : \neg\textbf{ac}(A)(s,t)) \\
\&\ (A, B) & : ((\textit{act\_Formula} \times \textit{act\_Formula}) \rightarrow \textit{act\_Formula}) \\
 & \stackrel{\text{def}}{=} \textbf{action}(\lambda(s, t : \textit{St}) : \textbf{ac}(A)(s,t) \wedge \textbf{ac}(B)(s,t))
\end{array}
$$

The disjunction (|) operator on actions is derived from the negation and conjunction operators. □

An action operator applied to formulae of type *act_Formula* will return an action formula (of type *act_Formula*) and a temporal logic operator will return a temporal formula (of type *tla_Formula*). Definitions of the nonstandard operators are given in Figure 2.

The quantifier for rigid variables is represented as a higher order function, **existc**, constructing a TLA formula with the **exf** form.

Definition 9 *Quantification over rigid variables*

The existential operator for rigid variables takes a function from a value to a TLA formula and returns a well-formed formula.

$$
\begin{array}{rl}
\textbf{existc}(H) & : ((\textit{Vals} \rightarrow \textit{tla\_Formula}) \rightarrow \textit{tla\_Formula}) \\
 & \stackrel{\text{def}}{=} \textbf{exf}(\lambda\sigma : \exists(c : \textit{Vals}) : \textbf{trans}(H(c), \sigma))
\end{array}
$$

□

The definition of the function **trans** gives the existential quantifier for rigid variables the interpretation $\mathbf{trans}(\mathbf{existc}(H), \sigma) = (\exists (c : \mathit{Vals}) : \mathbf{trans}(H(c), \sigma))$.

Existential quantification over program variables

The reduction function on behaviours, $\natural$, cannot be defined by primitive recursion: if the behaviour σ satisfies $\forall i : \exists j : \sigma_i(0) \neq \sigma_i(j)$ then $\natural\sigma$ never terminates. Instead, a function, *stut*, which specifies $\natural$ is used in the definition of the quantifier.

The definition of *stut* is based on the indices of the state changes in a behaviour σ where i is the index of a state change iff $\sigma(i) \neq \sigma(i+1)$. The first state change in a behaviour is the least index and is specified using a description operator (ϵ). The nth state change, $n > 0$, is found by dropping the prefix of σ up to and including the $(n-1)$th state change.

The epsilon operator, ϵ, returns any element of a given set; if the set is empty, ϵ returns any value. The function *choose* applies the epsilon operator to a given nonempty set[1]. The PVS types *set*$[T]$ and *nonempty?*$[T]$ define the sets of type T and nonempty sets of type T respectively.

Definition 10 *Choice*

The choice function *choose* takes a nonempty set S and returns some element of S.

$$\mathit{choose} : (\mathit{nonempty?}[T] \to T) \stackrel{\text{def}}{=} \epsilon(S)$$

where $\epsilon(S) \in S$ □

For a behaviour σ, the set $I(\sigma)$ contains the naturals which are indices of state changes in σ. The first state change in σ is contained in the, singleton, subset of $I(\sigma)$ given by *least*$(I(\sigma))$.

Definition 11 *Indices of state changes*

The function I takes a behaviour σ and returns the set of naturals such that $i \in I(\sigma) \Leftrightarrow \sigma(i) \neq \sigma(i+1)$.

$$\begin{aligned} I &: (\mathit{Behaviour} \to \mathit{set}[\mathit{nat}]) \\ I(\sigma) &\stackrel{\text{def}}{=} \{i : \mathit{nat} \mid \sigma(i) \neq \sigma(i+1)\} \end{aligned}$$

Given a set of naturals, *least* returns the subset containing the least element of the set.

$$\begin{aligned} \mathit{least} &: (\mathit{set}[\mathit{nat}] \to \mathit{set}[\mathit{nat}]) \\ \mathit{least}(S) &\stackrel{\text{def}}{=} \{i \mid i \in S \wedge \neg(\exists j : j \in S \wedge j < i)\} \end{aligned}$$

□

If a behaviour σ has any state change, the index of the first is given by *choose*(*least*$(I(\sigma))$) since *choose* returns any element of the (singleton) set *least*$(I(\sigma))$.

A behaviour which contains no state changes is said to be empty.

Definition 12 *Empty behaviours*

A behaviour is empty if every state in the behaviour is equal to the first state.

$$\mathit{empty?}(\sigma) \Leftrightarrow \forall n : \sigma(n) = \sigma(0)$$

□

[1] The ϵ operator and the function *choose* are specified in the PVS library.

Corollary 1 *For any behaviour σ, empty?$(\sigma) \Leftrightarrow I(\sigma) = \{\}$*

Proof. *empty?*$(\sigma) \Rightarrow I(\sigma) = \{\}$: For every natural, i, $\sigma(i) = \sigma(0) = \sigma(i+1)$ and $I(\sigma)$ is empty. $I(\sigma) = \{\} \Rightarrow$ *empty?*(σ): There is no index i of σ such that $\sigma(i) \neq \sigma(i)$ and *empty?*(σ) follows. □

The index of the nth state change in a behaviour σ is the index of the first state change after dropping the prefix of σ up-to and including the $(n-1)$th state change. This gives an index relative to the $(n-1)$th state change and the number of states in the prefix must be added to get the actual index.

Definition 13 *Index of a state change in a behaviour*
The function *next* takes a behaviour σ and a natural n and returns the index of the nth state change in σ.

$$\begin{array}{rl} \mathit{next} & : ((\mathit{Behaviour} \times \mathit{nat}) \to \mathit{nat}) \\ \mathit{next}(\sigma, n) & \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \mathit{empty?}(\sigma) \\ \mathit{choose}(\mathit{least}(I(\sigma))) & \text{if } n = 0 \\ \mathit{next}(\sigma, n-1) + 1 + \mathit{next}(\sigma_{\mathit{next}(\sigma,n-1)+1}, 0) & \text{otherwise} \end{cases} \end{array}$$

□

The function *next* terminates since the number of self-references is bounded by the argument n. The function *stut*, on a behaviour σ returns a behaviour β constructed from the states of σ which differ from their successor and satisfying $\beta(n) = \sigma(\mathit{next}(\sigma, n))$.

Definition 14 *Reduction function*
The function *stut* specifies the reduction function on behaviours.

$$\begin{array}{rl} \mathit{stut} & : (\mathit{Behaviour} \to \mathit{Behaviour}) \\ \mathit{stut}(\sigma) & \stackrel{\text{def}}{=} (\lambda(n : \mathit{nat}) : \sigma(\mathit{next}(\sigma, n))) \end{array}$$

□

The function *stut* is equivalent to the TLA behaviour reduction function.

Theorem 1 *For any behaviour σ, stut$(\sigma) = \natural\sigma$.*

Proof. The proof is given in Appendix A and is by extensionality and induction with three main steps:

1. a proof that $\mathit{stut}(\sigma)(0) = \natural\sigma(0)$,
2. a proof that $(\natural\sigma)_n = \natural(\sigma_{\mathit{next}(\sigma,n)})$ and
3. with $\mathit{stut}(\sigma)(n)$ written as $\sigma(\mathit{next}(\sigma, n))$ and $\natural\sigma(n)$ as $(\natural\sigma)_n(0)$, using the results of (1) and (2) to show that $\sigma(\mathit{next}(\sigma, n)) = \natural(\sigma(\mathit{next}(\sigma, n)))(0)$.

□

As well as the reduction function, existential quantification over program variables uses the equality over behaviours, $=_x$, and this is defined by the predicate *equals?*.

Definition 15 *Existential quantifier*

The function *equals?* has type $((\mathit{Vars} \rightarrow (\mathit{Behaviour} \times \mathit{Behaviour})) \rightarrow \mathit{boolean})$ and is defined as

$$\mathit{equals?}(v)(\sigma, \beta) \stackrel{\text{def}}{=} (\forall(w : \mathit{Vars}) : (\forall(n : \mathit{nat}) : w \neq v \Rightarrow \sigma(n)(w) = \beta(n)(w)))$$

The existential operator on program variables, **existv**, takes a variable and a function from the variables to the TLA formulae and returns a well formed formula.

$$\begin{array}{ll} \multicolumn{2}{l}{\mathbf{existv}(v, F) : ((\mathit{Vars} \times (\mathit{Vars} \rightarrow \mathit{tla\_Formula}) \rightarrow \mathit{tla\_Formula}))} \\ \stackrel{\text{def}}{=} & \mathbf{exf}(\lambda\sigma : (\exists(\beta, \tau : \mathit{Behaviour}) : \quad \mathit{stut}(\sigma) = \mathit{stut}(\beta) \wedge \mathit{equals?}(v)(\beta, \tau) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge \mathbf{trans}(F(v), \tau))) \end{array}$$

□

5 Example

A simple example of the representation of a TLA formula is taken from Lamport (1994). A TLA formula Φ specifies a program with two integer variables which are both initially 0 and whose values increase in steps.

$$\begin{array}{l} \mathit{init} \stackrel{\text{def}}{=} (x = 0) \wedge (y = 0) \\ M_1 \stackrel{\text{def}}{=} (x' = x + 1) \wedge (y' = y) \quad M_2 \stackrel{\text{def}}{=} (y' = y + 1) \wedge (x' = x) \\ M \stackrel{\text{def}}{=} M_1 \vee M_2 \\ \Phi \stackrel{\text{def}}{=} \mathit{init} \wedge \Box [M]_{(x,y)} \wedge \mathrm{WF}_{(x,y)}(M_1) \wedge \mathrm{WF}_{(x,y)}(M_2) \end{array}$$

init is a predicate, M_1, M_2 and M are actions and Φ is a temporal formula. The identifiers and values are defined as types.

$$\mathit{Vars}\text{: } \mathtt{type} = \{x, y\} \quad \mathit{Vals}\text{: } \mathtt{type} = \mathit{int}$$

The predicate *init* contains only unprimed variables and is translated to a predicate on a state. It is defined as a lambda expression taking a state s and each occurrence of x and y in *init* is replaced with $s(x)$ and $s(y)$ respectively. The **action** constructor returns an action formula.

$$\mathbf{init} : \mathit{act\_Formula} \stackrel{\text{def}}{=} \mathbf{action}(\mathbf{unprimed}(\lambda(s : \mathit{St}) : (s(x) = 0) \wedge (s(y) = 0)))$$

The actions M_1 and M_2 are represented in the same way: they are defined as lambda expressions taking a pair of states, s and t; each unprimed variable v is replaced by $s(v)$ and each primed variable v' is replaced by $t(v)$.

$$\begin{array}{l} \mathbf{m}_1 : \mathit{act\_Formula} \stackrel{\text{def}}{=} \mathbf{action}(\lambda(s, t : \mathit{St}) : (t(x) = (s(x) + 1)) \wedge (t(y) = s(y))) \\ \mathbf{m}_2 : \mathit{act\_Formula} \stackrel{\text{def}}{=} \mathbf{action}(\lambda(s, t : \mathit{St}) : (t(y) = (s(y) + 1)) \wedge (t(x) = s(x))) \end{array}$$

The action M is a disjunction of actions and can be translated as the disjunction of either temporal or action formulae. The first gives an instance of the TLA formulae type and disallows the use of **m** where an action is required. The second returns an action which may be used in both temporal and action formulae.

$$\mathbf{m} : \mathit{act\_Formula} \stackrel{\text{def}}{=} \mathbf{m}_1 \mid \mathbf{m}_2$$

The formula Φ has terms which take a tuple of the program variables, (x, y), as an argument. The value of the tuple is given by the value of the variables in a particular state and the tuple can be considered a function taking variables and returning a state function. The two-place tuple is defined to take a PVS tuple of variables and returns a function from a state to a PVS tuple of values.

$$\begin{aligned}\textbf{tuple}(a, b) \quad &: ((\mathit{Vars} \times \mathit{Vars}) \to (\mathit{St} \to (\mathit{Vals} \times \mathit{Vals}))) \\ &\stackrel{\text{def}}{=} (\lambda(s : State) : (s(a), s(b)))\end{aligned}$$

The TLA formula Φ is translated as

$$\begin{aligned}\textbf{Phi} : \mathit{tla\_Formula} \stackrel{\text{def}}{=} \; &\textbf{init and square}(\mathbf{m}, \textbf{tuple}(x, y)) \\ &\textbf{and WF}(\mathbf{m}_1, \textbf{tuple}(x, y)) \textbf{ and WF}(\mathbf{m}_2, \textbf{tuple}(x, y))\end{aligned}$$

This combines the actions and predicates using the propositional operators defined on the type *tla_Formula* to return an element of type *tla_Formula.*

6 Conclusion

The model used in the implementation follows that of the TLA report (Lamport, 1994) and is based on infinite sequences of states in which the semantics of the operators are defined. The use of infinite sequences to represent behaviours means that the TLA function for reducing stuttering behaviours, $\natural$ cannot be given directly. Instead, a function *stut* is defined which specifies, and is equivalent to, the reduction function. It is this specification function which is used in the definition of the existential quantifier. The model is similar to that used by von Wright & Långbacka for the implementation on HOL, in which the states are represented as tuples of values and the behaviours as sequences of state. In the implementation for Isabelle (Kalvala, 1995), a behaviour is represented as a list of the changes made to states with the empty list representing the behaviour in which there are no state changes.

The formulae of TLA are defined as instances of a type of well-formed formula, as the subset of this type containing only the actions and as the PVS representation of actions. These simplify the definition of the TLA syntax since only a small number of basic operators are needed. The separation of temporal and action formulae is common in other implementations: in the HOL implementation of TLA, actions are lambda expressions on tuples of values and temporal formulae are lambda expressions on behaviours. In the Isabelle implementation, actions are predicates on states, the syntax of TLA is defined as an abstract data type and the interpretation of a TLA formulae is as a predicate on behaviours. In both of these implementations, separate operators are defined for action and temporal formulae.

References

Abadi, M. (1993). An axiomatisation of Lamport's temporal logic of actions. Technical Report 65, DEC Systems Research Center.

Busch, H. (1995). A practical method for reasoning about distributed systems in a theorem prover. In Schubert, E. T. et al. (Eds.), *HUG '95*, Volume 971 of *Lecture Notes in Computer Science*. Springer-Verlag.

Engberg, U. (1995). *Reasoning in the Temporal Logic of Actions*. Ph. D. thesis, Department of Computer Science, University of Aarhus.

Engberg, U., Grønning, P. and Lamport, L. (1992). Mechanical verification of concurrent systems with TLA. In von Bochmann, G. and Probst, D. K. (Eds.), *Proceeding of the 4th International Workshop on Computer Aided Verification*, Volume 663 of *Lecture Notes in Computer Science*. Springer-Verlag.

Garland, S. J. and Guttag, J. V. (1991). A guide to LP, the Larch Prover. Technical Report 82, DEC Systems Research Center.

Kalvala, S. (1995). A formulation of TLA in Isabelle. Available from the Computer Laboratory, University of Cambridge.

Lamport, L. (1994, may). The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems 16*(3), 872–923.

Långbacka, T. (1994). A HOL formalisation of the Temporal Logic of Actions. In Melham, T. F. and Camilleri, J. (Eds.), *Higher Order Logic Theorem Proving and Its Applications*, Volume 859 of *Lecture Notes in Computer Science*, pp. 332–354. Springer-Verlag.

Melham, T. F. (1993). *Higher Order Logic and Hardware Verification*. Cambridge Tracts in Theoretical Computer Science. Cambridge University.

Owre, S., Rushby, J. M. and Shankar, N. (1993). PVS: A Prototype Verification System. Technical Report SRI-CSL-93-04, SRI.

Rescher, N. and Urquahart, A. (1971). *Temporal Logic*. Library of Exact Philosophy. Springer-Verlag/Wien.

von Wright, J. and Långbacka, T. (1992). Using a theorem prover for reasoning about concurrent algorithms. In *CAV '92*, Volume 663 of *Lecture Notes in Computer Science*. Springer-Verlag.

A Stuttering Behaviours

The proof was carried out using the PVS proof checker with the definition of the function $\natural$ given as an axiom and used as a rewrite rule.

Two properties of the suffix function are used: for any behaviour σ and naturals n and m, $\sigma_{(n+m)} = (\sigma_n)_m$ and $\sigma_n(0) = \sigma(n)$. Both can be proved from the definition of the suffix.

A.1 Properties of *choose* and *next*

For a nonempty behaviour, σ, any member of the set returned by *least*$(I(\sigma))$ is the index of the first state change of σ.

Lemma 1 *Properties of choose: I*
For any behaviour σ which is not empty, $\neg$empty?(σ),

$$\forall n : n < \textit{choose}(\textit{least}(I(\sigma))) \Rightarrow \sigma(n) = \sigma(n-1)$$

Proof. Since $\neg$*empty?*(σ), $I(\sigma)$ is not empty and there is a unique natural $i \in I(\sigma)$ which is less than any other member of $I(\sigma)$. From the definitions, *least*$(I(\sigma)) = \{i\}$ and *choose*$(\{i\}) = \epsilon\{i\} = i$ and $\sigma(i) \neq \sigma(i+1)$. Since $i \in$ *least*$(I(\sigma))$ it follows that $\forall n : n < i \Rightarrow \sigma(n) = \sigma(n+1)$ (otherwise $n \in I(\sigma)$ contradicting $i \in$ *least*$(I(\sigma))$). □

A consequence of Lemma 1 and the definition of $\natural$ is that reducing any behaviour, σ, is equivalent to reducing any suffix of that behaviour up to the first state change of σ.

Lemma 2 *Properties of choose: II*
For any behaviour σ which is not empty, $\neg$empty?(σ),

$$\forall i : i \leq \mathit{choose}(\mathit{least}(I(\sigma))) \Rightarrow \natural(\sigma_i) = \natural(\sigma)$$

Proof. The proof uses the formula

$$\forall i < \mathit{choose}(\mathit{least}(I(\sigma))) : \natural(\sigma_i) = \natural(\sigma_{i+1}) \tag{1}$$

Proof of Formula 1:

Since $i < \mathit{choose}(\mathit{least}(I(\sigma)))$ and $\mathit{least}(I(\sigma))$ is a singleton set, $i \notin \mathit{least}(I(\sigma))$ and $i \notin I(\sigma)$ giving $\sigma(i) = \sigma(i+1)$ and $\sigma_i(0) = \sigma_i(1)$. By definition $\natural(\sigma_i) = \natural(\sigma_{i+1})$.

Proof of Lemma 2: By induction on $i \leq \mathit{choose}(\mathit{least}(I(\sigma)))$.

Case $i = 0$: $\natural(\sigma_0) = \natural(\sigma_0)$ is trivially true.

Case $i > 0$ and $\natural(\sigma_0) = \natural(\sigma_{(i-1)})$: From Formula 1, $\natural(\sigma_{i-1}) = \natural(\sigma_i)$ and from the assumption, $\natural(\sigma_0) = \natural(\sigma_i)$. □

For any nonempty behaviour σ, the state indexed by $\mathit{next}(\sigma, 0)$ differs from its successor.

Lemma 3 *Properties of next*
For any behaviour σ which is not empty,

$$\sigma(\mathit{next}(\sigma, 0)) \neq \sigma(\mathit{next}(\sigma, 0) + 1)$$

Proof. From the assumption that σ is not empty and from the definition of *next*,

$$\mathit{next}(\sigma, 0) = \mathit{choose}(\mathit{least}(I(\sigma)))$$

From the epsilon axiom,

$$\begin{array}{lll}
 & \mathit{choose}(\mathit{least}(I(\sigma))) \in \mathit{least}(I(\sigma)) & \\
\Rightarrow & \mathit{choose}(\mathit{least}(I(\sigma))) \in I(\sigma) & \text{(definition of } \mathit{least}\text{)} \\
\Rightarrow & \sigma(\mathit{choose}(\mathit{least}(I(\sigma)))) \neq \sigma(\mathit{choose}(\mathit{least}(I(\sigma))) + 1) & \text{(definition of } I\text{)} \\
\Rightarrow & \sigma(\mathit{next}(\sigma, 0)) \neq \sigma(\mathit{next}(\sigma, 0) + 1) & \text{(definition of } \mathit{next}\text{)}
\end{array}$$

□

A.2 Equivalence of *stut*(σ) and $\natural(\sigma)$

For empty behaviours the proof is immediate.

Lemma 4 *Equivalence for empty behaviours*
For any empty behaviour σ, stut(σ) and $\natural\sigma$ are equivalent.

$$\mathit{empty?}(\sigma) \Rightarrow \forall n : \mathit{stut}(\sigma)(n) = \natural\sigma(n)$$

Proof. The definition of $\natural$ gives $\natural\sigma = \sigma$ and $\natural\sigma(n) = \sigma(n)$. The definition of *next* gives $\mathit{next}(\sigma, n) = 0$ and $\mathit{stut}(\sigma)(n) = \sigma(\mathit{next}(\sigma, n)) = \sigma(0)$. From the definition of $\mathit{empty?}(\sigma)$, $\forall n : \sigma(n) = \sigma(0)$.

□

The proof of equivalence for nonempty behaviours uses the fact that the behaviours $\mathit{stut}(\sigma)$ and $\natural\sigma$ have the same first state.

Theorem 2 *Base case*
For any behaviour σ, $\mathit{stut}(\sigma)(0) = \natural\sigma(0)$

Proof. The case when σ is empty is proved by Lemma 4.

Assuming σ is not empty, the definition of *stut* gives $\sigma(\mathit{next}(\sigma, 0)) = \natural\sigma(0)$ and from the definition of *next*,

$$\begin{array}{lll} & \mathit{next}(\sigma,0) = \mathit{choose}(\mathit{least}(I(\sigma))) & \\ \Rightarrow & \natural\sigma = \natural(\sigma_{\mathit{next}(\sigma,0)}) & \text{(Lemma 2)} \\ \Rightarrow & \sigma(\mathit{next}(\sigma,0)) \neq \sigma(\mathit{next}(\sigma,0)+1) & \text{(Lemma 3)} \\ \Rightarrow & \natural(\sigma_{\mathit{next}(\sigma,0)})(0) = \sigma(\mathit{next}(\sigma,0)) & \text{(definition of } \natural\text{)} \end{array}$$

□

The proof of the general case is based on the the equivalence $(\natural\sigma)_n = \natural(\sigma_{\mathit{next}(\sigma,n)})$. This, in turn, uses a relationship between the reduction of the suffix of a behaviour starting at the nth state and the reduction of the suffix starting at the $n+1$th state change.

Lemma 5 *For any behaviour σ,*

$$\natural(\sigma_{\mathit{next}(\sigma,n+1)}) = \natural(\sigma_{(\mathit{next}(\sigma,n)+1)})$$

Proof. With $N = \mathit{next}(\sigma, n)$, the definition of *next* gives

$$\mathit{next}(\sigma, n+1) = (N+1) + \mathit{next}(\sigma_{(N+1)}, 0)$$

Case empty?(σ_N): If σ_N is is empty then so is $\sigma_{(N+1)}$ and $\mathit{next}(\sigma, n+1) = (N+1)$. Substituting for N gives $\mathit{next}(\sigma, n+1) = \mathit{next}(\sigma, n) + 1$ and $\natural(\sigma_{\mathit{next}(\sigma,n+1)}) = \natural(\sigma_{\mathit{next}(\sigma,n)+1})$.

Case $\neg$*empty?*(σ_N) and *empty?*(σ_{N+1}): If σ_N is not empty but $\sigma_{(N+1)}$ is then $\mathit{next}(\sigma_{(N+1)}, 0) = 0$ and $\mathit{next}(\sigma, n+1) = (N+1)$. Substituting for N gives $\mathit{next}(\sigma, n+1) = \mathit{next}(\sigma, n) + 1$ and $\natural(\sigma_{\mathit{next}(\sigma,n+1)}) = \natural(\sigma_{\mathit{next}(\sigma,n)+1})$.

Case $\neg$*empty?*(σ_N) and $\neg$*empty?*(σ_{N+1}):

With $\beta = \sigma_{(\mathit{next}(\sigma,n)+1)}$ $(= \sigma_{(N+1)})$,

$$\mathit{next}(\sigma, n+1) = (N+1) + \mathit{next}(\beta, 0)$$

which gives

$$\begin{array}{lll} & \natural\beta = \natural(\sigma_{((N+1)+\mathit{next}(\beta,0))}) & \\ \Rightarrow & \natural((\sigma_{(N+1)})_{\mathit{next}(\beta,0)}) = \natural(\beta_{\mathit{next}(\beta,0)}) & \text{(property of suffix)} \\ \Rightarrow & \natural(\beta_{\mathit{next}(\beta,0)}) = \natural\beta & (\neg\mathit{empty?}(\beta)\text{, definition of } \mathit{next} \text{ and Lemma 2)} \end{array}$$

Substituting for β and N gives,

$$\begin{aligned} \natural(\sigma_{(\mathit{next}(\sigma,n)+1)}) &= \natural(\sigma_{(\mathit{next}(\sigma,n)+1+\mathit{next}(\beta,0))}) \\ &= \natural(\sigma_{(\mathit{next}(\sigma,n)+1+\mathit{next}(\sigma_{(\mathit{next}(\sigma,n)+1)},0))}) \\ &= \natural(\sigma_{\mathit{next}(\sigma,n+1)}) \end{aligned}$$

□

The function *next* defines a relationship between the suffix of a behaviour and the suffix of the reduced behaviour.

Theorem 3 *For any behaviour σ and natural n,*

$$(\natural\sigma)_n = \natural(\sigma_{\mathit{next}(\sigma,n)})$$

Proof. If σ is empty, $\mathit{next}(\sigma, n) = 0$, $\natural(\sigma_0) = \sigma_0$ and $(\natural\sigma)_n = \sigma_n$. By extensionality, $\sigma_0 = \sigma_n$. Assuming that σ is not empty, the proof is by induction on n.

Case $n = 0$: From the definition of $\mathit{next}(\sigma, 0)$ and Lemma 2, $(\natural\sigma) = \natural(\sigma_{\mathit{next}(\sigma,0)})$.

Case $n > 0$ and assuming $(\natural\sigma)_{(n-1)} = \natural(\sigma_{\mathit{next}(\sigma,n-1)})$:

With $N = \mathit{next}(\sigma, n-1)$ and $\beta = \sigma_{(\mathit{next}(\sigma,n-1)+1)}$ the definition of *next* gives

$$\mathit{next}(\sigma, n) = (N+1) + \mathit{next}(\sigma_{(N+1)}, 0)$$

The value of $\mathit{next}(\sigma, n)$ depends on whether σ_N is empty.

Case empty?(σ_N): If σ_N is empty then so is every suffix of σ_N including $\sigma_{(N+1)}$.

$$\begin{array}{lll} & \natural(\sigma_{(N+1)}) = \sigma_{(N+1)} & \\ \Rightarrow & \natural(\sigma_{(\mathit{next}(\sigma,n-1)+1)}) = \sigma_{\mathit{next}(\sigma,n-1)+1} & \text{(substituting for } N\text{)} \\ \Rightarrow & \natural(\sigma_{\mathit{next}(\sigma,n)}) = \sigma_{\mathit{next}(\sigma,n-1)+1} & \text{(Lemma 5)} \end{array}$$

From the inductive hypothesis,

$$\begin{array}{lll} & (\natural\sigma)_{n-1} = \natural(\sigma_{\mathit{next}(\sigma,n-1)}) & \\ \Rightarrow & \natural(\sigma_{\mathit{next}(\sigma,n-1)}) = \sigma_{\mathit{next}(\sigma,n-1)} & (\sigma_N \text{ is empty and } N = \mathit{next}(\sigma, n-1)) \\ & \sigma_{\mathit{next}(\sigma,n-1)} = (\sigma_{\mathit{next}(\sigma,n-1)})_1 & (\sigma_{\mathit{next}(\sigma,n-1)} \text{ is empty}) \\ \Rightarrow & (\natural\sigma)_n = (\sigma_{\mathit{next}(\sigma,n-1)})_1 & \\ \Rightarrow & (\natural\sigma)_n = \sigma_{\mathit{next}(\sigma,n-1)+1} & \text{(property of suffixes)} \end{array}$$

Case $\neg$*empty?*(σ_N): From the property of suffixes,

$$\begin{array}{lll} & (\natural\sigma)_n = ((\natural\sigma)_{(n-1)})_1 & \\ \Rightarrow & (\natural\sigma)_n = (\natural(\sigma_N))_1 & \text{(inductive hypothesis)} \\ \Rightarrow & (\natural\sigma)_n = (\natural(\sigma_{\mathit{next}(\sigma,n-1)}))_1 & \text{(replacing } N\text{)} \end{array}$$

By extensionality, for any natural c,

$$\begin{array}{lll} & (\natural\sigma)_n(c) = (\natural(\sigma_N))_1(c) & \\ \Rightarrow & (\natural\sigma)_n(c) = \natural(\sigma_N)(c+1) & \text{(definition of suffix)} \\ & \sigma_N(0) \neq \sigma_N(1) & \text{(Lemma 3)} \\ \Rightarrow & (\natural\sigma)_n(c) = \natural(\sigma_{N+1})(c) & \text{(definition of } \natural\text{)} \\ \Rightarrow & (\natural\sigma)_n(c) = \natural(\sigma_{\mathit{next}(\sigma,n-1)+1})(c) & \text{(replacing } N\text{)} \\ \Rightarrow & (\natural\sigma)_n(c) = \natural(\sigma_{\mathit{next}(\sigma,n)})(c) & \text{(Lemma 5)} \\ \Rightarrow & (\natural\sigma)_n = \natural(\sigma_{\mathit{next}(\sigma,n)}) & \text{(extensionality)} \end{array}$$

□

The proof of equivalence follows from Theorems 3 and 2.

Theorem 4 *General case*
For any behaviour σ, stut$(\sigma) = \natural\sigma$.

Proof. By extensionality, for any natural n, $\mathit{stut}(\sigma)(n) = \natural\sigma(n)$. If $n = 0$, the equivalence is proved by Theorem 2. Assume that $n > 0$.

By definition,

$$\begin{aligned} \mathit{stut}(\sigma)(n) &= \sigma(\mathit{next}(\sigma, n)) \\ &= \sigma_{\mathit{next}(\sigma,n)}(0) \quad \text{(definition of suffix)} \end{aligned}$$

Also from the definition of suffix,

$$\begin{aligned} \natural\sigma(n) &= (\natural\sigma)_n(0) \\ &= \natural(\sigma_{\mathit{next}(\sigma,n)})(0) && \text{(Theorem 3)} \\ &= \mathit{stut}(\sigma_{\mathit{next}(\sigma,n)})(0) && \text{(Theorem 2)} \\ &= \sigma_{\mathit{next}(\sigma,n)}(\mathit{next}(\sigma_{\mathit{next}(\sigma,n)}, 0)) && \text{(definition of } \mathit{stut}) \end{aligned}$$

The equivalence to be proved is therefore

$$\sigma_{\mathit{next}(\sigma,n)}(0) = \sigma_{\mathit{next}(\sigma,n)}(\mathit{next}(\sigma_{\mathit{next}(\sigma,n)}, 0))$$

By definition, $\mathit{next}(\sigma, n) = \mathit{next}(\sigma, n-1)+1+\mathit{next}(\sigma_{(\mathit{next}(\sigma,n-1)+1)}, 0)$. With $\beta = \sigma_{(\mathit{next}(\sigma,n-1)+1)}$, the proof is by cases of *empty?*(β).

Case empty?$(\sigma_{(\mathit{next}(\sigma,n-1)+1)})$:

From the definition of *next*,

$$\begin{aligned} & \mathit{next}(\sigma_{(\mathit{next}(\sigma,n-1)+1)}, 0) = 0 && \text{(definition of } \mathit{next}) \\ \Rightarrow\ & \mathit{next}(\sigma, n) = (\mathit{next}(\sigma, n-1) + 1) \\ \Rightarrow\ & \sigma_{\mathit{next}(\sigma,n)}(0) = \sigma_{(\mathit{next}(\sigma,n-1)+1)}(0) \end{aligned}$$

This leads to

$$\begin{aligned} & \sigma_{\mathit{next}(\sigma,n)}(\mathit{next}(\sigma_{\mathit{next}(\sigma,n)}, 0)) \\ &= \sigma_{\mathit{next}(\sigma,n-1)+1}(\mathit{next}(\sigma_{\mathit{next}(\sigma,n-1)+1}, 0)) \\ &= \sigma_{\mathit{next}(\sigma,n)}(0) && (\sigma_{\mathit{next}(\sigma,n-1)+1} \text{ is empty}) \end{aligned}$$

Case $\neg$empty?$(\sigma_{(\mathit{next}(\sigma,n-1)+1)})$:

For $\natural\sigma(n) = \sigma_{\mathit{next}(\sigma,n)}(\mathit{next}(\sigma_{\mathit{next}(\sigma,n)}, 0))$, the proof is by showing $\mathit{next}(\sigma_{\mathit{next}(\sigma,n)}, 0) = 0$.

The definition of *next* gives

$$\begin{aligned} & \mathit{next}(\sigma_{\mathit{next}(\sigma,n)}, 0) \\ &= \mathit{next}(\sigma_{(\mathit{next}(\sigma,n-1)+1+\mathit{next}(\beta,0))}, 0) && (\beta = \sigma_{\mathit{next}(\sigma,n-1)+1}) \\ &= \mathit{next}((\sigma_{(\mathit{next}(\sigma,n-1)+1)})_{\mathit{next}(\beta,0)}, 0) && \text{(writing with a suffix)} \end{aligned}$$

Writing this with β gives

$$\mathit{next}(\sigma_{\mathit{next}(\sigma,n)}, 0) = \mathit{next}(\beta_{\mathit{next}(\beta,0)}, 0) \tag{2}$$

β is not empty and from Lemma 3, $\beta(\mathit{next}(\beta, 0)) \neq \beta(\mathit{next}(\beta, 0) + 1)$.

$$
\begin{array}{lll}
 & \beta(\mathit{next}(\beta, 0)) = \beta_{\mathit{next}(\beta,0)}(0) & \\
\text{and} & \beta(\mathit{next}(\beta, 0) + 1) = \beta_{\mathit{next}(\beta,0)}(1) & \\
\Rightarrow & \mathit{next}(\beta, 0) \in I(\beta_{\mathit{next}(\beta,0)}) & \text{(definition of } I\text{)} \\
\text{and} & 0 \in \mathit{least}(I(\beta_{\mathit{next}(\beta,0)})) & (\beta_{\mathit{next}(\beta,0)}(0) \neq \beta_{\mathit{next}(\beta,0)}(1)) \\
\Rightarrow & \mathit{next}(\beta_{\mathit{next}(\beta,0)}, 0) = \mathit{choose}(\mathit{least}(I(\beta_{\mathit{next}(\beta,0)}))) & \text{(definition of } \mathit{next}\text{)} \\
\Rightarrow & \mathit{next}(\beta_{\mathit{next}(\beta,0)}, 0) = 0 & \\
\Rightarrow & \mathit{next}(\sigma_{\mathit{next}(\sigma,n)}, 0) = 0 & \text{(Formula 2)}
\end{array}
$$

$\sigma_{\mathit{next}(\sigma,n)}(\mathit{next}(\sigma_{\mathit{next}(\sigma,n)}, 0))$ is therefore $\sigma_{\mathit{next}(\sigma,n)}(0)$ and

$$\natural\sigma(n) = \sigma_{\mathit{next}(\sigma,n)}(0) = \mathit{stut}(\sigma)(n)$$

completing the proof.

$\square$