Interval Temporal Logic

Antonio Cau and Ben Moszkowski

2019-01-02

Abstract

Interval Temporal Logic (ITL) is a flexible notation for both propositional and first-order reasoning about periods of time found in descriptions of hardware and software systems. Unlike most temporal logics, ITL can handle both sequential and parallel composition and offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time [129]. Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulas in a slightly modified version of ITL [120]. Tempura provides an executable framework for developing and experimenting with suitable ITL specifications. In addition, ITL and its mature executable subset Tempura [152] have been extensively used to specify the properties of real-time systems where the primitive circuits can directly be represented by a set of simple temporal formulae. In addition, Tempura has been applied to hardware simulation and other areas where timing is important.
# Contents

1 Finite Interval Temporal Logic 3  
1.1 Syntax ......................................................... 3  
1.2 Semantics .................................................... 3  
1.3 Derived Constructs ......................................... 5  
1.4 Propositional proof system ................................. 7  
1.5 First order proof system .................................... 7

2 Finite and Infinite Interval Temporal Logic 8  
2.1 Syntax .......................................................... 8  
2.2 Semantics ....................................................... 8  
2.3 Derived Constructs .......................................... 11  
2.4 Propositional proof system ................................. 13  
2.5 First order proof system .................................... 13

3 Tools 14  
3.1 (Ana)Tempura .................................................. 14  
3.2 FLCheck: Fusion Logic decision Procedure ................. 21  
3.3 ITL library for Isabelle/HOL ................................. 23  
3.4 ITL Theorem Prover based on Prover9 ...................... 23  
3.5 ITL Proof Checker based on PVS ........................... 24  
3.6 Automatic Verification of Interval Temporal Logic ...... 24

4 ITL Related Publications 25  
4.1 Articles ......................................................... 25  
4.2 In Collections ................................................ 29  
4.3 Conference Papers ............................................ 31  
4.4 Books .......................................................... 39  
4.5 Theses .......................................................... 40  
4.6 Technical Reports ............................................ 41
1 Finite Interval Temporal Logic

The key notion of ITL is an interval. An interval $\sigma$ is considered to be a finite sequence of states $\sigma_0 \ldots \sigma_k$.

1.1 Syntax

The syntax of finite ITL is defined in Table 1 where
- $z$ denotes an integer value,
- $a$ denotes a static integer variable,
- $A$ denotes a state integer variable,
- $b$ denotes a Boolean value,
- $q$ denotes a static propositional variable,
- $Q$ denotes a state propositional variable,
- ig denotes a integer function symbol,
- bg denotes a Boolean function symbol,
- $v$ denotes static or state (integer or Boolean) variable,
- $e_i$ denotes a Boolean or integer expression,
- $h$ denotes a predicate symbol.

1.2 Semantics

Each state $\sigma_i$ is the union of the mapping from the set of integer variables $\text{IntVar}$ to the set of integer values $\mathbb{Z}$ and the mapping from propositional variables $\text{PropVar}$ to set of Boolean values $\{\text{tt}, \text{ff}\}$.

Each interval has at least one state. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to $n$, one less than the number of states in the interval (this has always been a convention in ITL), i.e., a one state interval has length 0. Let $\sigma = \sigma_0 \sigma_1 \sigma_2 \ldots$ be an interval then
- $\sigma_0 \ldots \sigma_k$ (where $0 \leq k \leq |\sigma|$) denotes a prefix interval of $\sigma$
- $\sigma_k \ldots \sigma_{|\sigma|}$ (where $0 \leq k \leq |\sigma|$) denotes a suffix interval of $\sigma$
- $\sigma_k \ldots \sigma_l$ (where $0 \leq k \leq l \leq |\sigma|$) denotes a sub interval of $\sigma$

The informal semantics of the most interesting constructs are as follows:
- $\bigcirc A$: if the interval is non-empty then the value of $A$ in the next state of that interval else an arbitrary value.
- $\text{fin } A$: the value of $A$ in the last state of the interval.
- $\text{skip}$ unit interval (length 1).
Table 2: Semantics of finite ITL

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E<a href="%5Csigma">z</a>$</td>
<td>$z$</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">a</a>$</td>
<td>$\sigma_0(a)$ and for all $0 &lt; i \leq</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">A</a>$</td>
<td>$\sigma_0(A)$</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">\text{ig}(i e_1, \ldots, i e_n)</a>$</td>
<td>$\text{ig}(E<a href="%5Csigma">e_1</a>, \ldots, E<a href="%5Csigma">e_n</a>)$</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">\bigcirc A</a>$</td>
<td>$\sigma_1(A)$ if $</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">\text{fin } A</a>$</td>
<td>$\sigma_{</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">b</a>$</td>
<td>$b$</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">q</a>$</td>
<td>$\sigma_0(q)$ and for all $0 &lt; i \leq</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">Q</a>$</td>
<td>$\sigma_0(Q)$</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">\text{bg}(b e_1, \ldots, b e_n)</a>$</td>
<td>$\text{bg}(E<a href="%5Csigma">e_1</a>, \ldots, E<a href="%5Csigma">e_n</a>)$</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">\bigcirc Q</a>$</td>
<td>$\sigma_1(Q)$ if $</td>
</tr>
<tr>
<td>$E<a href="%5Csigma">\text{fin } Q</a>$</td>
<td>$\sigma_{</td>
</tr>
<tr>
<td>$M<a href="%5Csigma">\text{true}</a>$</td>
<td>$\text{tt}$</td>
</tr>
<tr>
<td>$M<a href="%5Csigma">h(e_1, \ldots, e_n)</a>$</td>
<td>$\text{tt}$ iff $h(E<a href="%5Csigma">e_1</a>, \ldots, E<a href="%5Csigma">e_n</a>)$</td>
</tr>
<tr>
<td>$M<a href="%5Csigma">\neg f</a>$</td>
<td>$\text{tt}$ iff $\neg (M<a href="%5Csigma">f</a> = \text{tt})$</td>
</tr>
<tr>
<td>$M<a href="%5Csigma">f_1 \land f_2</a>$</td>
<td>$\text{tt}$ iff $(M<a href="%5Csigma">f_1</a> = \text{tt}) \land (M<a href="%5Csigma">f_2</a> = \text{tt})$</td>
</tr>
<tr>
<td>$M<a href="%5Csigma">\neg \forall v \cdot f</a>$</td>
<td>$\text{tt}$ iff $</td>
</tr>
<tr>
<td>$M<a href="%5Csigma">f_1 ; f_2</a>$</td>
<td>$\text{tt}$ iff $(\exists k, \text{s.t. } M[f_1](\sigma_0 \ldots \sigma_k) = \text{tt}) \land (M[f_2](\sigma_k \ldots \sigma_{</td>
</tr>
<tr>
<td>$M<a href="%5Csigma">\exists f^*</a>$</td>
<td>$\text{tt}$ iff $(\exists l_0, \ldots, l_n, \text{s.t. } l_0 = 0 \land l_n =</td>
</tr>
</tbody>
</table>

- $f_1 ; f_2$ holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that $f_1$ holds over the prefix interval and $f_2$ over the suffix interval.
- $f^*$ holds if the interval is decomposable into a finite number of intervals such that for each of them $f$ holds.

Let $\Sigma^+$ denote the set of all finite intervals.
Let Expressions denote the set of (integer or Boolean) expressions.
Let Val denote the set of integer or Boolean values ($\mathbb{Z} \cup \text{Bool}$).
Let $E[\ldots](\ )$ denote the meaning function from Expressions $\times \Sigma^+$ to Val.
Let Formulae denote the set of ITL formulae.
Let $M[\ldots](\ )$ denote the meaning function from Formulae $\times \Sigma^+$ to Bool (set of Boolean values, $\{\text{tt}, \text{ff}\}$).
Let $\sigma = \sigma_0 \sigma_1 \ldots$ denote an interval.
We write $\sigma \sim_v \sigma'$ if the intervals $\sigma$ and $\sigma'$ are identical with the possible exception of their mappings for the variable $v$.
Let choose-any-from(Val) denote the choice function that selects an arbitrary value from Val.
The formal semantics is listed in Table 2:
1.3 Derived Constructs
Frequently used derived constructs are listed in Table 3–6.

Table 3: Frequently used non-temporal derived constructs
<table>
<thead>
<tr>
<th>Formula</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg \text{true} )</td>
<td>( \neg f ) &lt;true &gt; or ( \neg f )</td>
</tr>
<tr>
<td>( \neg (\neg f \land \neg f) )</td>
<td>( f_1 \lor f_2 )</td>
</tr>
<tr>
<td>( \neg f_1 \lor \neg f_2 )</td>
<td>( f_1 \supset f_2 )</td>
</tr>
<tr>
<td>( f_1 \equiv f_2 )</td>
<td>( f_1 \equiv f_2 \equiv (f_1 \supset f_2) \land (f_2 \supset f_1) )</td>
</tr>
<tr>
<td>( \exists v \bullet f )</td>
<td>( \exists v \bullet f \equiv \neg \forall v \bullet \neg f )</td>
</tr>
</tbody>
</table>

Table 4: Frequently used temporal derived constructs
<table>
<thead>
<tr>
<th>Formula</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \circ f )</td>
<td>( \circ f \equiv \text{skip} ; f )</td>
</tr>
<tr>
<td>more</td>
<td>( \circ \text{true} )</td>
</tr>
<tr>
<td>empty</td>
<td>( \circ \neg f )</td>
</tr>
<tr>
<td>( \diamond f \equiv \text{true} ; f )</td>
<td>( \diamond f \equiv \text{true} ; f )</td>
</tr>
<tr>
<td>( \Box f \equiv \neg (\diamond \neg f) )</td>
<td>( \Box f \equiv \neg (\diamond \neg f) )</td>
</tr>
<tr>
<td>( \Diamond f \equiv f ; \neg f )</td>
<td>( \Diamond f \equiv f ; \neg f )</td>
</tr>
<tr>
<td>( \Box (f) \equiv \text{empty} \lor (\Box (f) ; \text{skip}) )</td>
<td>( \Box (f) \equiv \text{empty} \lor (\Box (f) ; \text{skip}) )</td>
</tr>
<tr>
<td>( \Diamond (f) \equiv \neg (\Box (f) ; \text{skip}) )</td>
<td>( \Diamond (f) \equiv \neg (\Box (f) ; \text{skip}) )</td>
</tr>
<tr>
<td>( \Diamond (f) \equiv f \land \Box (\neg f) )</td>
<td>( \Diamond (f) \equiv f \land \Box (\neg f) )</td>
</tr>
</tbody>
</table>
Table 5: Frequently used concrete derived constructs

<table>
<thead>
<tr>
<th>Construct</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( f_0 ) then ( f_1 ) else ( f_2 )</td>
<td>( (f_0 \land f_1) \lor (\neg f_0 \land f_2) ) if then else</td>
</tr>
<tr>
<td>if ( f_0 ) then ( f_1 ) else true</td>
<td>if ( f_0 ) then ( f_1 ) else true if then</td>
</tr>
<tr>
<td>fin ( f )</td>
<td>( \square(\text{empty} \supset f) ) final state</td>
</tr>
<tr>
<td>halt ( f )</td>
<td>( \square(\text{empty} \equiv f) ) terminate interval when</td>
</tr>
<tr>
<td>keep ( f )</td>
<td>( \Diamond(\text{skip} \supset f) ) all unit subintervals</td>
</tr>
<tr>
<td>keepnow ( f )</td>
<td>( \Diamond^*(\text{skip} \land f) ) initial unit subinterval</td>
</tr>
<tr>
<td>while ( f_0 ) do ( f_1 )</td>
<td>( f_0 \land (\text{while} \neg f_1 \text{ do } f_0) ) while loop</td>
</tr>
<tr>
<td>repeat ( f_0 ) until ( f_1 )</td>
<td>( f_0 ; (\text{while} \neg f_1 \text{ do } f_0) ) repeat loop</td>
</tr>
<tr>
<td>( f_1 \mapsto f_0 )</td>
<td>( \Diamond(f_1 \supset f_0) ) always followed by</td>
</tr>
<tr>
<td>( f_1 \leftrightarrow f_0 )</td>
<td>( \Diamond(f_1 \equiv f_0) ) strong followed by</td>
</tr>
</tbody>
</table>

Table 6: Frequently used derived constructs related to expressions

<table>
<thead>
<tr>
<th>Construct</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Y := e )</td>
<td>( (\Diamond Y) = e ) assignment</td>
</tr>
<tr>
<td>( Y \approx e )</td>
<td>( \square(Y = e) ) equal in interval</td>
</tr>
<tr>
<td>( Y \leftarrow e )</td>
<td>( (\text{fin } Y) = e ) temporal assignment</td>
</tr>
<tr>
<td>( Y \text{ gets } e )</td>
<td>( \text{keep } (Y \leftarrow e) ) gets</td>
</tr>
<tr>
<td>stable ( Y )</td>
<td>( Y \text{ gets } Y ) stability</td>
</tr>
<tr>
<td>padded ( Y )</td>
<td>( \text{padded } (Y \leftarrow e) \lor \text{empty} ) padded expression</td>
</tr>
<tr>
<td>( Y &lt;\sim e )</td>
<td>( (Y \leftarrow e) \land \text{padded } Y ) padded temporal assignment</td>
</tr>
<tr>
<td>intlen ( (e) )</td>
<td>( \exists I \cdot (I = 0) \land (I \text{ gets } I + 1) \land (I \leftarrow e) ) interval length</td>
</tr>
</tbody>
</table>
1.4 Propositional proof system

In Table 7 we list the propositional axioms and rules for finite ITL.

<table>
<thead>
<tr>
<th>Table 7: Propositional Axioms and Rules for finite ITL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChopAssoc ( \vdash (f_0 ; f_1) ; f_2 \equiv f_0 ; (f_1 ; f_2) )</td>
</tr>
<tr>
<td>OrChopImp ( \vdash (f_0 \lor f_1) ; f_2 \supset (f_0 ; f_2) \lor (f_1 ; f_2) )</td>
</tr>
<tr>
<td>ChopOrImp ( \vdash f_0 ; (f_1 \lor f_2) \supset (f_0 ; f_1) \lor (f_0 ; f_2) )</td>
</tr>
<tr>
<td>EmptyChop ( \vdash \text{empty} ; f \equiv f )</td>
</tr>
<tr>
<td>ChopEmpty ( \vdash f ; \text{empty} \equiv f )</td>
</tr>
<tr>
<td>BiBoxChopImpChop ( \vdash 2i (f_0 \supset f_1) \land 2i (f_2 \supset f_3) \supset (f_0 ; f_2) \supset (f_1 ; f_3) )</td>
</tr>
<tr>
<td>StateImpBi ( \vdash p \supset \square p )</td>
</tr>
<tr>
<td>NextImpNotNextNot ( \vdash \square f \supset \neg \neg \square f )</td>
</tr>
<tr>
<td>BoxInduct ( \vdash f_0 \land 2i (f_0 \supset \square w f) \supset 2i f_0 )</td>
</tr>
<tr>
<td>ChopStarEqv ( \vdash f^* \equiv (\text{empty} \lor ((f \land \text{more}) ; f^*)) )</td>
</tr>
<tr>
<td>MP ( \vdash f_0 \supset f_1, \vdash f_0 \Rightarrow \vdash f_1 )</td>
</tr>
<tr>
<td>BoxGen ( \vdash f_0 \Rightarrow \vdash \square f_0 )</td>
</tr>
<tr>
<td>BiGen ( \vdash f_0 \Rightarrow \vdash \square f_0 )</td>
</tr>
</tbody>
</table>

1.5 First order proof system

Some axioms for the first order case are shown in Table 8.

Let \( \nu \) refer to both static and state variables.

We denote by \( f[e / \nu] \) that in formula \( f \) expression \( e \) is substituted for variable \( \nu \).

<table>
<thead>
<tr>
<th>Table 8: Some First Order Axioms and Rules for finite ITL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForallSub ( \vdash \forall \nu \cdot f \supset f[e / \nu] ), where the expression ( e ) has the same data and temporal type as the variable ( \nu ) and is free for ( \nu ) in ( f ).</td>
</tr>
<tr>
<td>ForallImplies ( \vdash \forall \nu \cdot (f_1 \supset f_2) \supset (f_1 \supset \forall \nu \cdot f_2) ), where ( \nu ) doesn’t occur freely in ( f_1 ).</td>
</tr>
<tr>
<td>SubstAxiom ( \vdash \square (v_1 = v_2) \supset f \equiv f [v_2 / v_1] ).</td>
</tr>
<tr>
<td>StaticWeakNext ( \vdash w \supset \Diamond w ), where ( w ) only contains static variables.</td>
</tr>
<tr>
<td>ExistsChopRight ( \vdash \exists \nu \cdot (f_1 ; f_2) \supset (\exists \nu \cdot f_1) ; f_2 ), where ( \nu ) doesn’t occur freely in ( f_2 ).</td>
</tr>
<tr>
<td>ExistsChopLeft ( \vdash \exists \nu \cdot (f_1 ; f_2) \supset f_1 ; (\exists \nu \cdot f_2) ), where ( \nu ) doesn’t occur freely in ( f_1 ).</td>
</tr>
<tr>
<td>ForallGen ( \vdash f \Rightarrow \vdash \forall \nu \cdot f ), for any variable ( \nu ).</td>
</tr>
</tbody>
</table>
Table 9: Syntax of finite and infinite ITL

Expressions
\[ ie ::= z \mid a \mid A \mid ig(i_1, \ldots, i_n) \mid \circ A \mid \text{fin } A \]

Boolean Expressions
\[ be ::= b \mid q \mid Q \mid bg(b_1, \ldots, b_n) \mid \circ Q \mid \text{fin } Q \]

Formulae
\[ f ::= \text{true} \mid h(e_1, \ldots, e_n) \mid \neg f \mid f_1 \land f_2 \mid \forall v \cdot f \mid \text{skip} \mid f_1 ; f_2 \mid f^* \]

2 Finite and Infinite Interval Temporal Logic

The key notion of ITL is an interval. An interval \( \sigma \) is considered to be a (in)finite sequence of states \( \sigma_0 \sigma_1 \ldots \).

2.1 Syntax

The syntax of ITL is defined in Table 9 where
- \( z \) denotes an integer value,
- \( a \) denotes a static integer variable,
- \( A \) denotes a state integer variable,
- \( b \) denotes a Boolean value,
- \( q \) denotes a static propositional variable,
- \( Q \) denotes a state propositional variable,
- \( ig \) denotes an integer function symbol,
- \( bg \) denotes a Boolean function symbol,
- \( v \) denotes static or state integer variable,
- \( e_i \) denotes a Boolean or integer expression,
- \( h \) denotes a predicate symbol.

2.2 Semantics

Each state \( \sigma_i \) is the union of the mapping from the set of integer variables \( \text{IntVar} \) to the set of integer values \( \mathbb{Z} \) and the mapping from propositional variables \( \text{PropVar} \) to set of Boolean values \{tt, ff\}.

Each interval has at least one state. The length \( |\sigma| \) of an interval \( \sigma_0 \ldots \sigma_n \) is equal to \( n \), one less than the number of states in the interval (this has always been a convention in ITL), i.e., a one state interval has length 0. Let \( \sigma = \sigma_0 \sigma_1 \sigma_2 \ldots \) be an interval then

- \( \sigma_0 \ldots \sigma_k \) (where \( 0 \leq k \leq |\sigma| \)) denotes a prefix interval of \( \sigma \)
- \( \sigma_k \ldots \sigma_{|\sigma|} \) (where \( 0 \leq k \leq |\sigma| \)) denotes a suffix interval of \( \sigma \)
- \( \sigma_k \ldots \sigma_l \) (where \( 0 \leq k \leq l \leq |\sigma| \)) denotes a sub interval of \( \sigma \)

The informal semantics of the most interesting constructs are as follows:

- \( \circ A \): if interval is non-empty then the value of \( A \) in the next state of that interval else an arbitrary value.
- \( \text{fin } A \): if interval is finite then the value of \( A \) in the last state of that interval else an arbitrary value.
• skip unit interval (length 1).

• \( f_1 ; f_2 \) holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that \( f_1 \) holds over the prefix and \( f_2 \) over the suffix, or if the interval is infinite and \( f_2 \) holds for that interval.

• \( f^* \) holds if the interval is decomposable into a finite number of intervals such that for each of them \( f \) holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which \( f \) holds.

Let \( \Sigma^+ \) denote the set of all finite intervals and \( \Sigma^\omega \) denotes the set of all infinite intervals.

Let Expressions denote the set of (integer or Boolean) expressions.

Let Val denote the set of integer or Boolean values \( (\mathbb{Z} \cup \text{Bool}) \).

Let \( E[\ldots]() \) denote the meaning function from Expressions \( \times (\Sigma^+ \cup \Sigma^\omega) \) to Val.

Let Formulae denote the set of ITL formulae.

Let \( M[\ldots]() \) denote the meaning function from Formulae \( \times (\Sigma^+ \cup \Sigma^\omega) \) to Bool (set of Boolean values, \( \{\text{tt}, \text{ff}\} \)).

Let \( \sigma = \sigma_0 \sigma_1 \ldots \) denote an interval.

We write \( \sigma \sim_v \sigma' \) if the intervals \( \sigma \) and \( \sigma' \) are identical with the possible exception of their mappings for the variable \( v \).

Let choose-any-from(Val) denote the choice function that selects an arbitrary value from Val.

The formal semantics is listed in Table 10:
Table 10: Semantics of finite and infinite ITL

\[
\begin{align*}
E[z](\sigma) &= z \\
E[a](\sigma) &= \sigma_0(a) \text{ and for all } 0 < i \leq |\sigma|, \sigma_i(a) = \sigma_0(a) \\
E[A](\sigma) &= \sigma_0(A) \\
E[i_1\ldots, i_n](\sigma) &= ig(E[i_1](\sigma), \ldots, E[i_n](\sigma)) \\
E[\circ A](\sigma) &= \sigma_1(A) \text{ (if } |\sigma| > 0 \text{)} \\
&\quad \text{choose-any-from}(\mathbb{Z}) \text{ otherwise} \\
E[\text{fin } A](\sigma) &= \sigma_{|\sigma|}(A) \text{ (if } \sigma \text{ is finite)} \\
&\quad \text{choose-any-from}(\mathbb{Z}) \text{ otherwise} \\
E[b](\sigma) &= b \\
E[q](\sigma) &= \sigma_0(q) \text{ and for all } 0 < i \leq |\sigma|, \sigma_i(q) = \sigma_0(q) \\
E[Q](\sigma) &= \sigma_0(Q) \\
E[bg](\sigma) &= bg(E[\sigma_1](\sigma), \ldots, E[\sigma_n](\sigma)) \\
E[\circ Q](\sigma) &= \sigma_1(Q) \text{ (if } |\sigma| > 0 \text{)} \\
&\quad \text{choose-any-from}(\mathbb{Z}) \text{ otherwise} \\
E[\text{fin } Q](\sigma) &= \sigma_{|\sigma|}(Q) \text{ (if } \sigma \text{ is finite)} \\
&\quad \text{choose-any-from}(\mathbb{Z}) \text{ otherwise} \\
M[\text{true}](\sigma) &= \text{tt} \\
M[h](\sigma) &= \text{tt (iff } h(E[e_1](\sigma), \ldots, E[e_n](\sigma)) \text{)} \\
M[\neg f](\sigma) &= \text{tt (iff not } (M[f](\sigma) = \text{tt}) \text{)} \\
M[f_1 \land f_2](\sigma) &= \text{tt (iff } (M[f_1](\sigma) = \text{tt}) \text{ and } (M[f_2](\sigma) = \text{tt}) \text{)} \\
M[\text{skip}](\sigma) &= \text{tt (iff } |\sigma| = 1 \text{)} \\
M[\forall v \bullet f](\sigma) &= \text{tt (iff } (\text{for all } \sigma' \text{ s.t. } \sigma \sim_v \sigma', M[f](\sigma') = \text{tt}) \text{)} \\
M[f_1 ; f_2](\sigma) &= \text{tt (if } (\text{exists } k, \text{ s.t. } M[f_1](\sigma_0 \ldots \sigma_k) = \text{tt and } M[f_2](\sigma_k \ldots \sigma_{|\sigma|}) = \text{tt}) \text{)} \\
&\quad \text{or } (\sigma \text{ is infinite and } M[f_1](\sigma) = \text{tt}) \text{)} \\
M[f^*](\sigma) &= \text{tt (iff } (\text{if } \sigma \text{ is finite then)} \\
&\quad (\text{exist } l_0, \ldots, l_n \text{ s.t. } l_0 = 0 \text{ and } l_n = |\sigma| \text{ and)} \\
&\qquad \text{for all } 0 \leq i < n, l_i < l_{i+1} \text{ and } M[f](\sigma_{l_i} \ldots \sigma_{l_{i+1}}) = \text{tt}) \text{)} \\
&\quad \text{else)} \\
&\quad (\text{exist } l_0, \ldots, l_n \text{ s.t. } l_0 = 0 \text{ and)} \\
&\quad \quad M[f](\sigma_{l_0} \ldots \sigma_{|\sigma|}) = \text{tt and} \\
&\quad \quad \text{for all } 0 \leq i < n, l_i < l_{i+1} \text{ and } M[f](\sigma_{l_i} \ldots \sigma_{l_{i+1}}) = \text{tt}) \text{)} \\
&\quad \text{or)} \\
&\quad (\text{exist an infinite number of } l_i \text{ s.t. } l_0 = 0 \text{ and)} \\
&\quad \quad \text{for all } 0 \leq i, l_i < l_{i+1} \text{ and } M[f](\sigma_{l_i} \ldots \sigma_{l_{i+1}}) = \text{tt}) \text{)}
\end{align*}
\]
### 2.3 Derived Constructs

Frequently used derived constructs are listed in Table 11–14.

<table>
<thead>
<tr>
<th>Table 11: Frequently used non-temporal derived constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>False ( \triangleq \neg \text{true} )</td>
</tr>
<tr>
<td>( f_1 \lor f_2 \triangleq \neg(\neg f_1 \land \neg f_2) )</td>
</tr>
<tr>
<td>( f_1 \supset f_2 \triangleq \neg f_1 \lor f_2 )</td>
</tr>
<tr>
<td>( f_1 \equiv f_2 \triangleq (f_1 \supset f_2) \land (f_2 \supset f_1) )</td>
</tr>
<tr>
<td>( \exists v \bullet f \triangleq \neg \forall v \bullet \neg f )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 12: Frequently used temporal derived constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Diamond f \triangleq \text{skip} ; f )</td>
</tr>
<tr>
<td>more ( \triangleq \Diamond \text{true} )</td>
</tr>
<tr>
<td>empty ( \triangleq \neg \text{more} )</td>
</tr>
<tr>
<td>inf ( \triangleq \text{true} ; \neg \text{false} )</td>
</tr>
<tr>
<td>( \text{isinf}(f) \triangleq \text{inf} \land f )</td>
</tr>
<tr>
<td>finite ( \triangleq \neg \text{inf} )</td>
</tr>
<tr>
<td>( \text{isfin}(f) \triangleq \text{finite} \land f )</td>
</tr>
<tr>
<td>fmore ( \triangleq \text{more} \land \text{finite} )</td>
</tr>
<tr>
<td>( \Diamond f \triangleq \text{finite} ; f )</td>
</tr>
<tr>
<td>( \square f \triangleq \neg \Diamond \neg f )</td>
</tr>
<tr>
<td>( \Diamond f \triangleq \neg \Diamond f )</td>
</tr>
<tr>
<td>( \Diamond f \triangleq f ; \text{true} )</td>
</tr>
<tr>
<td>( \square f \triangleq \neg(\Diamond \neg f) )</td>
</tr>
<tr>
<td>( \Diamond f \triangleq \text{finite} ; f ; \text{true} )</td>
</tr>
<tr>
<td>( \square f \triangleq \neg(\Diamond \neg f) )</td>
</tr>
</tbody>
</table>
Table 13: Frequently used concrete derived constructs

<table>
<thead>
<tr>
<th>Construct</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $f_0$ then $f_1$ else $f_2$</td>
<td>$\equiv (f_0 \land f_1) \lor (\neg f_0 \land f_2)$</td>
</tr>
<tr>
<td>if $f_0$ then $f_1$</td>
<td>$\equiv$ if then $f_0$ then $f_1$ else true</td>
</tr>
<tr>
<td>$\text{fin } f$</td>
<td>$\equiv$ $\square(\text{empty } \supset f)$</td>
</tr>
<tr>
<td>$\text{sfin } f$</td>
<td>$\equiv$ $\neg(\text{fin } (\neg f))$</td>
</tr>
<tr>
<td>$\text{halt } f$</td>
<td>$\equiv$ $\square(\text{empty } \equiv f)$</td>
</tr>
<tr>
<td>$\text{shalt } f$</td>
<td>$\equiv$ $\neg(\text{halt } (\neg f))$</td>
</tr>
<tr>
<td>$\text{keep } f$</td>
<td>$\equiv$ $\square(\text{skip } \supset f)$</td>
</tr>
<tr>
<td>$f^\omega$</td>
<td>$\equiv \text{isinf } (\text{isfin } (f)^\omega)$</td>
</tr>
<tr>
<td>$f^* (f)$</td>
<td>$\equiv \text{isfin } (\text{isfin } (f)^<em>) \lor$ $\text{isfin } (\text{isfin } (f)^</em>) \land \text{isinf } (f)$</td>
</tr>
<tr>
<td>while $f_0$ do $f_1$</td>
<td>$\equiv (f_0 \land f_1)^* \land \text{fin } \neg f_0$</td>
</tr>
<tr>
<td>repeat $f_0$ until $f_1$</td>
<td>$\equiv f_0 ; (\text{while } \neg f_1 \text{ do } f_0)$</td>
</tr>
</tbody>
</table>

Table 14: Frequently used derived constructs related to expressions

<table>
<thead>
<tr>
<th>Construct</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y := e$</td>
<td>$\equiv (\triangledown Y) = e$</td>
</tr>
<tr>
<td>$Y \approx e$</td>
<td>$\equiv \Box (Y = e)$</td>
</tr>
<tr>
<td>$Y \leftarrow e$</td>
<td>$\equiv$ finite $\land (\text{fin } Y) = e$</td>
</tr>
<tr>
<td>$Y$ gets $e$</td>
<td>$\equiv$ keep $(Y \leftarrow e)$</td>
</tr>
<tr>
<td>stable $Y$</td>
<td>$\equiv Y$ gets $Y$</td>
</tr>
<tr>
<td>padded $Y$</td>
<td>$\equiv (\text{stable } (Y) ; \text{skip}) \lor \text{empty}$</td>
</tr>
<tr>
<td>$Y &lt;\sim e$</td>
<td>$\equiv (Y \leftarrow e) \land \text{padded } Y$</td>
</tr>
<tr>
<td>intlen $(e)$</td>
<td>$\equiv \exists I \cdot (I = 0) \land (I$ gets $I + 1) \land (I \leftarrow e)$</td>
</tr>
</tbody>
</table>
2.4 Propositional proof system

In Table 15 we list the propositional axioms and rules for finite and infinite ITL.

Table 15: Propositional Axioms and Rules for finite and infinite ITL.

- **ChopAssoc**: \(\vdash (f_0 ; f_1) ; f_2 \equiv f_0 ; (f_1 ; f_2)\)
- **OrChopImp**: \(\vdash (f_0 \lor f_2) ; f_2 \supset (f_0 ; f_2) \lor (f_1 ; f_2)\)
- **ChopOrImp**: \(\vdash f_0 ; (f_1 \lor f_2) \supset (f_0 ; f_1) \lor (f_0 ; f_2)\)
- **EmptyChop**: \(\vdash \text{empty} ; f \equiv f\)
- **ChopEmpty**: \(\vdash f ; \text{empty} \equiv f\)
- **BiBoxChopImpChop**: \(\vdash 2i(f_0 \supset f_1) \land 2i(f_2 \supset f_3) \supset (f_0 ; f_2) \supset (f_1 ; f_3)\)
- **StateImpBi**: \(\vdash p \supset 2i p\)
- **NextImpNotNextNot**: \(\vdash \Box f \supset \neg \Diamond \neg f\)
- **BoxInduct**: \(\vdash f \land 2i(f \supset \Box w f) \supset 2i f\)
- **InfChop**: \(\vdash (f \land \inf) ; g \equiv (f \land \inf)\)
- **ChopStarEqv**: \(\vdash f \equiv (\text{empty} \lor ((f \land \text{more}) ; f^*)\)
- **MP**: \(\vdash f_0 \supset f_1, \vdash f_0 \Rightarrow \vdash f_1\)

2.5 First order proof system

Some axioms for the first order case are shown in Table 16. Let \(v\) refer to both static and state variables.

We denote by \(f [e / v]\) that in formula \(f\) expression \(e\) is substituted for variable \(v\).

Table 16: Some First Order Axioms and Rules for ITL.

- **ForallSub**: \(\vdash \forall v \cdot f \supset f [e / v]\), where the expression \(e\) has the same data and temporal type as the variable \(v\) and is free for \(v\) in \(f\).
- **ForallImplies**: \(\vdash \forall v \cdot (f_1 \supset f_2) \supset (f_1 \supset \forall v \cdot f_2)\), where \(v\) doesn’t occur freely in \(f_1\).
- **SubstAxiom**: \(\vdash \Box (v_1 = v_2) \supset f \equiv f [v_2 / v_1]\).
- **StaticWeakNext**: \(\vdash w \supset \Box w\), where \(w\) only contains static variables.
- **ExistsChopRight**: \(\vdash \exists v \cdot (f_1 ; f_2) \supset (\exists v \cdot f_1) ; f_2\), where \(v\) doesn’t occur freely in \(f_2\).
- **ExistsChopLeft**: \(\vdash \exists v \cdot (f_1 ; f_2) \supset f_1 ; (\exists v \cdot f_2)\), where \(v\) doesn’t occur freely in \(f_1\).
- **ForallGen**: \(\vdash f \Rightarrow \vdash \forall v \cdot f\), for any variable \(v\).
3 Tools

3.1 (Ana)Tempura

Tempura, the C-Tempura interpreter version 2.7 developed originally by Roger Hale and now maintained by Antonio Cau and Ben Moszkowski, is an interpreter for executable Interval Temporal Logic formulae. The first Tempura interpreter was programmed in Prolog by Ben Moszkowski, and was operational around December 2, 1983. Subsequently he rewrote the interpreter in Lisp (mid Mar, 1984), and in late 1984 modified the program to handle a two-level memory and multi-pass scanning. The C-Tempura interpreter was written in early 1985 by Roger Hale at Cambridge University.

AnaTempura, which is built upon C-Tempura, is a tool for the runtime verification of systems using Interval Temporal Logic (ITL) and its executable subset Tempura. The runtime verification technique uses assertion points to check whether a system satisfies timing, safety or security properties expressed in ITL. The assertion points are inserted in the source code of the system and will generate a sequence of information (system states), like values of variables and timestamps of value change, while the system is running. Since an ITL property corresponds to a set of sequences of states (intervals), runtime verification is just checking whether the sequence generated by the system is a member of the set of sequences corresponding to the property we want to check. The Tempura interpreter is used to do this membership test.

Download Version:

- Version 3.4 (released 19/12/2017) gzipped tar file or zip file.
  - Initial support for monitoring rmi based Java programs.
  - Use internal variable runid to assign id to external processes.
  - Tempura commands stable and output now allows lists of variables, i.e., stable(V,W) and output(V,W).
  - Use kitcreator to generate the anatempura binaries. The windows binaries are generated using the mingw-w64 cross-compiler.
  - Various other bug fixes, see ChangeLog for more details.

- Version 3.3 (released 07/06/2016) gzipped tar file or zip file.
  - The Tcl/Tk graphical user interface does not depend on Expect anymore.
  - The lexer/parser now throws an error on encountering a unknown character instead of silently discarding it.
  - The read-only State_number variable can be used in Tempura programs to determine the current state number.
  - The monitoring of C# programs does not need a Java wrapper anymore.
  - Support for monitoring of Java programs in the form of a jar file.
  - Support of time-stamps in the form of seconds and microseconds.
  - The GUI has now a different look/layout, history of commands window is gone, the most commonly used menu entries are now buttons and one interacts with Tempura using a shell-like interface.
  - Added template .anatempurarc.
  - Integers are now 64bits regardless of the machine architecture.
  - Windows binaries are compiled using the msys2-mingw32/64 system.
- For Windows we have 32 bit (XP and beyond) and 64 bit (7 and beyond) Tempura/AnaTempura binaries.
- Random numbers are now generated using xorshift128plus.
- Added support for monitoring Scala programs.
- Added option -stdio to start anatempura in cmdline mode.
- Various other bug fixes, see ChangeLog for more details.

- Version 3.2 (released 30/11/2015) gzipped tar file or zip file.

- Rewritten the Tempura output capture/processing routine.
- Rewritten the external program data capture/processing routine.
- New implementation of memory/framed variables, existsf V : f, is now used to indicate that V is memory/framed variable within f. The previous syntax mem(V) is now deprecated.
- Fix bugs in the implementation of prev(L) where L is a list.
- Added Tempura binary for Arduino-Yun and Raspberry Pi, i.e., tempura_mips_openwrt_linux and tempura_arm_linux_gnueabihf.
- Added elapsed time in the statistics output at the end of a run.
- Added plc and sql injection detection examples.
- Various other bug fixes see ChangeLog for more details.

- Version 3.1 (released 22/05/2015) gzipped tar file or zip file.

- Changed contact email address.
- Added pre-compiled MacOSX binary.
- Added frame(V) as alias for mem(V).
- Fixed some bugs in implementation of mem(V).
- Fixed some bugs in the help command.

- Version 3.0 (released 04/07/2013) gzipped tar file or zip file.

- Dropped pre-compiled Solaris binary.
- flex/bison based parser backward compatible with previous hard coded version but with stricter syntactic checks
- changed from cvs to svn as version control system
- improved syntax error messages
- fixed memory leaks
- tidy up format command
- startup file .anatempurarc can also be in the current directory
- use kbs-0.4.4 to generate anatempura binaries
- anatempura binaries are using Tcl/Tk 8.6
- works again under Windows XP
- program assertions can have any symbol except control characters and !

- Version 2.18 (released 01/11/2012): gzipped tar file or zip file.
- Dropped tempura_macosx binary but added tempura_linux64 and anatempura_linux64 binaries.
- fixed some small bugs
- fixed memory leaks
- added command ‘winput’ that will wait for input from a file instead of switching to the keyboard.

  - initial support for MACOSX
  - fixed gui bugs
  - fixed some tempura bugs

- Version 2.16 (released 08/12/2009): gzipped tar file or zip file.
  - added floats. Floats have the form $2.3e+10$ in Tempura. For output: output($2.3$) will be $2.30000e+00$, i.e., precision is 5 digits after the ’.’. One can set this via the precision variable. With precision of 2 one gets $2.30e+00$. The format command can output floats in two forms: %f output will be of the form 2.33333, e output will be of the form 2.33333e+01. The following operations on floats are defined: unary, +, -; binary: +, -, div, mod, /, *, **, ceil, floor, sqrt, itof, exp, log, log10, sin, cos, tan, asin, acos, atan, atan2, sinh, cosh, tanh, fabs.
  - anatempura is now using the new Tile interface
  - when setting system variables with set, output both old and new values
  - Added 'frandom' and 'fRandom' for float random number between [0.0,1.0]
  - Added defaults command, X defaults 1 denotes when X is undefined then take as value for X the value 1.
  - Added prev(X) operator, the value of X in the previous state.
  - Added mem(X) operator, X is a ‘memory’ variable, i.e., when undefined take the value in the previous state.
  - Added #n history operator, used as option to exists when declaring a variable, it will keep a history of n previous values of a variable.
  - Added nprev(X,n) operator, nprev(X,3) for instance is an abbreviation of prev(prev(prev(X))).
  - When setting debug_level to 6 more useful information is displayed like the state of a variable and reduction rule being applied.
  - Included tempura executables tempura_linux for Linux (compiled on Ubuntu 9.10), tempura_solaris for Solaris (compiled on Sparc Solaris 10u8), and tempura.exe for Windows (compiled on Windows XP SP3).
  - Included anatempura executables anatempura_solaris, anatempura_linux and anatempura.exe. These were built using the Tclkit Kitgen build system (http://wiki.tcl.tk/18146). Now no need anymore to install tcl/tk and expect in order to run anatempura.
  - changed copyright license to GPLv3.0

- introduced various node accessor macros so that if one changes the node structure we only have to change the macro.
- if formula can’t be reduced in the final state of the prefix of a chop then we will evaluate ((prefix and empty);true) and (suffix). This feature can be switched on/off with hopchop. The default of hopchop is true.
- added integer overflow tests.
- unified/cleaned up the various node data structures.


- work around a recent misfeature of windows when started an external program.
- added the io redirections, set infile="some file name", set outfile="some file name", where stdin and stdout can be used to redirect to standard keyboard and screen i/o.
- added the infinite and randlen constructs for respectively an infinite interval and a random length interval (less or equal to max_randlen).


- added reset in file menu to restart tempura.
- open and reload now also load the file into Tempura.
- added showstate Tempura command. This will display what is (un)defined in the current state.
- changed contact email address to tempura@dmu.ac.uk

• Version 2.12 (released 04/05/2007): gzipped tar file or zip file.

This version is the first version that compiles both under Windows and Unix/Linux type of machines. See Changelog for detailed news/changes.

How to run AnaTempura?

• Use the pre-compiled binary:
  anatempura.exe: 32 bit binary and will run on Windows Xp, 7 and beyond, this will use tempura.exe in the current directory
  anatempura-win64.exe: 64 bit binary and will run on Windows 7 and beyond, this will use tempura-win64.exe in the current directory
  anatempura_linux: For 32bit Linux OS, this will use tempura_linux in the current directory
  anatempura_linux64: For 64bit Linux OS, this will use tempura_linux64 in the current directory
  anatempura_macosx: For MacOSX, this will use tempura_macosx in the current directory
• Use anatempura.tcl:
  you need to install Tempura, and Tcl/Tk (at least 8.5). Get Tcl/Tk and from Tcl/Tk site or use the ActiveTcl package.

  Tempura can be compiled using the Gnu C compiler under a Unix like operating system like Ubuntu, GNU Debian, etc. For Windows you can use the MSYS2/MinGW-32/64 system, see their website for downloading and installation. Install the MSYS2 system, and then, using pacman, install the mingw-w64-i686-toolchain for compiling 32bit binaries and the mingw-w64-x86_64-toolchain for compiling 64bit binaries.

  Compile Tempura using the following commands

```bash
./configure

make
```

  For convenience the following pre-compiled Tempura binaries are included in the distribution:

- 32 bit Windows binary tempura.exe,
- 64 bit Windows binary tempura-win64.exe,
- 32 bit Linux binary tempura_linux,
- 64 bit Linux binary tempura_linux64,
- MacOSX binary tempura_macosx,
- Arduino-Yun binary tempura_mips_openwrt_linux,
- Raspberry Pi binary tempura_arm_linux_gnuabihf.

Contact: Email cau.researcher@gmail.com in case of problems.

Publications:

• Analysing C programs is discussed in:

• Analysing Verilog programs is discussed in:

• A paper describing the run-time verification method used in AnaTempura:

• Slides of seminar talk about AnaTempura:

Overview: Figure 1 shows an overview of AnaTempura.
  Figure 2 shows the interface of AnaTempura.
  Figure 3 shows a graphical snapshot of a simulation of the EP/3 microprocessor specified in Tempura.
Figure 1: Overview of AnaTempura

Figure 2: Interface of AnaTempura
Figure 3: Graphical snapshot of simulation of the EP/3 microprocessor
3.2 FLCheck: Fusion Logic decision Procedure

Fusion Logic augments conventional Propositional Temporal Logic (PTL) with the fusion operator. Note: the fusion-operator is basically a “chop” that does not have an explicit negation on the left hand (for right fusion logic) side (as fusion expression) or the right hand (for left fusion logic). The negation is implicit, i.e., the negation is a derived fusion expression operator. The expressiveness of Fusion Logic is the same as Propositional Interval Temporal Logic. The main differences concern computational complexity, naturalness of expression for analytical purposes, and succinctness. Fusion Logic is closely related to Propositional Dynamic Logic (PDL).

We have implemented above decision procedure for Fusion Logic in Tcl/Tk and the CUDD BDD library. The tool allows one to check the validity or satisfiability of a Fusion Logic formula. If a formula is not valid it will produce a counter example and if a formula is satisfiable it will produce an example model. Figure 4 gives a screen dump of our tool which is available at

- **FLCHECK version 1.2** (released 22/05/2015).
  Main Changes:
  - Added support for MacOSX.
  - Added pre-compiled binary for MacOSX.

- **FLCHECK version 1.1** (released 04/07/2013).
  Main Changes:
  - Drop support for Solaris Sparc.
  - Added pre-compiled binaries for Windows (XP and 7) and Linux (Ubuntu 12.04, 32 and 64 bits).

- **FLCHECK version 1.0** (released 23/01/2013).
  Main Changes:
  - Simplified ‘left always followed by’ operator.
  - Added more examples.

- **FLCHECK version 0.9** (released 21/02/2012).
  Main Changes:
  - Introduction of left and right Fusion Logic which makes the specification of access control policies much simpler
  - Use of time reversal to rewrite left fusion logic formulae into right fusion logic formulae
  - Enforcement of policies expressed in left Fusion Logic
  - All examples come now with comments

- **FLCHECK version 0.8** (released 26/03/2010).
  Initial release.
Publications:


Figure 4: FLCHECK fusion logic decision procedure
3.3 ITL library for Isabelle/HOL

Isabelle/HOL is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus.

We have given a deep embedding of propositional ITL and a shallow embedding of first order ITL in Isabelle/HOL. A shallow embedding represents ITL using Isabelle/HOL predicates, while a deep embedding would represent ITL formulas as mutually inductive datatypes. See, e.g., [96] for a discussion about deep vs. shallow embeddings in Isabelle/HOL. The shallow embedding of first order ITL uses techniques developed by Stefan Merz [175, 20] for the shallow embedding of Temporal Logic of Actions (TLA) in Isabelle/HOL.

- **Deep embedding**: version 1.8 (08/12/2018, first public release), gzipped tar file.
  Contains:
  - Propositional ITL, syntax, semantics (finite intervals), and axioms and proof rules.
  - The lemmas from Imperative Reasoning in Interval Temporal Logic by Ben Moszkowski.
  - Time Reversal operator and an extensive list of lemmas.
  - First operator and Monitors. All key theorems/lemmas/semantics in David Smallwood PhD thesis “ITL Monitor: Compositional Runtime Analysis with Interval Temporal Logic” have been verified.
  - Link with Georg Struth’s work on Kleene Algebra. This work is based on Interval Temporal Algebra (Section 3.4) but now re-encoded in Isabelle/HOL. So the work on ITL and Prover9 and PVS has now been superseded by the Isabelle/HOL library.

- **Shallow embedding**: version 2.2 (08/12/2018, warning this is work in progress), gzipped tar file.
  Contains:
  - First Order ITL, both quantification over state and static variables, added temporal variables (current, next, final, penultimate), syntax, semantics, and axioms and proof rules. The technique used is similar to that of Stephan Merz [175, 20] used for encoding TLA in Isabelle/HOL.
  - The lemmas from Imperative Reasoning in Interval Temporal Logic by Ben Moszkowski.
  - Time Reversal operator and an extensive list of lemmas included. Time reversal operator also works for temporal variables and quantification.
  - First operator and Monitors. All key theorems/lemmas/semantics in David Smallwood PhD thesis “ITL Monitor: Compositional Runtime Analysis with Interval Temporal Logic” have been verified.
  - Some concrete monitors have been specified and properties verified.
  - Some examples using quantification.

3.4 ITL Theorem Prover based on Prover9

Note: this work has been re-encoded in the deep embedding of propositional ITL in Isabelle/HOL. Prover9 is a resolution/paramodulation automated theorem prover for first-order and equational logic developed by William McCune.
We have given an algebraic axiom system for Propositional Interval Temporal Logic (PITL): Interval Temporal Algebra. The axiom system is a combination of a variant of Kleene algebra and Omega algebra plus axioms for linearity and confluence.

This algebraic axiom system for PITL has been encoded in Prover9. So we can use Prover9 to prove the validity of various PITL theorems. The Prover9 encoding of PITL plus examples of more than 300 PITL theorems are available for download as:

  - documentation updated to new semantics for chopstar and chop omega algebraic operators
- Version 1.7 (released 15/05/2009): gzipped tar file.
  - updated documentation in doc, to use new ITL semantics
- Version 1.6 (released 12/12/2008): gzipped tar file.
  - changed copyright license to GPLv3.0 and added the notice to all files

The README in this tar file contains instructions how to use Prover9 for proving PITL theorems.

### 3.5 ITL Proof Checker based on PVS

Note: this work has been superseded by the deep and shallow embedding of ITL in Isabelle/HOL.

PVS is an interactive environment, developed at SRI, for writing formal specifications and checking formal proofs. The specification language used in PVS is a strongly typed higher order logic. The powerful interactive theorem prover/proof checker of PVS has a large set of basic deductive steps and the facility to combine these steps into proof strategies. PVS is implemented in Common Lisp —with ancillary functions provided in C, Tcl/TK and LaTeX— and uses GNU Emacs for its interface. PVS is freely available for IBM RS6000 machines as well as Sun Sparcs under license from SRI. See PVS homepage for more information.

- The ITL library for PVS 4.0.
- The ITL library for PVS 3.2.
- The ITL library for PVS 2.4 patchlevel 1.
- The ITL library for PVS 2.3.
- The ITL library for PVS 2.2.
- The ITL library for PVS 2.1 patchlevel 2.417.

**Publications:**

- Technical report.

### 3.6 Automatic Verification of Interval Temporal Logic

Shinji Kono has developed an automatic theorem prover for propositional ITL (LITE). The implementation is in Prolog. Further information can be gathered at Shinji Kono’s Interval Temporal Logic page. Shinji Kono has also a Java version of LITE see CVS repository of JavaLite.
4 ITL Related Publications

Download BibTeX file

The book Executing Temporal Logic Programs by Dr. B. C. Moszkowski was originally published by Cambridge University Press in 1986. The publishers have kindly given the copyright back to the author. The pdf version of the book has now been made available.

4.1 Articles


URL: http://dx.doi.org/10.1016/j.tcs.2012.01.026.

URL: http://dx.doi.org/10.1093/comjnl/bxs102.

URL: http://dx.doi.org/10.1016/j.ipl.2013.02.005.

URL: http://dx.doi.org/10.1016/j.tcs.2012.02.011.

URL: https://sites.google.com/site/ijcsis/vol-10-no-5-may-2012.


URL: https://sites.google.com/site/ijcsis/vol-9-no-8-aug-2011.

URL: http://dx.doi.org/10.1007/s00165-009-0130-y.


URL: http://dx.doi.org/10.1007/s11241-010-9094-5.


4.2 In Collections

URL: https://doi.org/10.1007/978-3-319-11313-5_38.

URL: http://dx.doi.org/10.1007/978-3-319-04921-2_36.

URL: http://dx.doi.org/10.1007/978-3-319-04915-1_3.

URL: http://dx.doi.org/10.1142/9789814271059_0008.

URL: http://dx.doi.org/10.1142/9789814271059_0005.
4.3 Conference Papers


URL: http://dx.doi.org/10.1007/978-3-319-25942-0_10.

URL: http://dx.doi.org/10.14236/ewic/ics-csr2014.9.

URL: http://dx.doi.org/10.1007/978-3-642-34281-3_20.

URL: http://dx.doi.org/10.1109/SCET.2012.6342060.

URL: https://doaj.org/article/373d1feae6e04b2c90b7457634ddfe4d.


URL: http://dx.doi.org/10.1007/978-3-642-34281-3_5.

URL: http://dx.doi.org/10.1109/CRiSIS.2011.6061544.

URL: http://dx.doi.org/10.1109/TIME.2011.25.

URL: http://dx.doi.org/10.1109/SSIRI.2010.7.


URL: http://dx.doi.org/10.1109/ICCD.1993.393373.

URL: http://dx.doi.org/10.1007/10.1109/1007-978-1-4471-3556-2_10.

URL: http://dx.doi.org/10.1109/EMWRT.1992.637480.


URL: http://dx.doi.org/10.1007/10.1109/3-540-56287-7_94.

URL: http://dx.doi.org/10.1007/10.1109/3-540-51803-7_35.


URL: http://dx.doi.org/10.1007/BFb0026305.

URL: http://dx.doi.org/10.1007/10.1107/3-540-17945-3_16.

URL: http://dx.doi.org/10.1007/10.1107/3-540-16479-0_24.


4.4 Books


4.5 Theses


4.6 Technical Reports

URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.422585.

URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.250763.

URL: http://www.cse.dmu.ac.uk/~jordan/root.pdf.

URL: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.393497.

URL: https://www.dora.dmu.ac.uk/handle/2086/4214.


URL: https://theses.ncl.ac.uk/dspace/handle/10443/2075.


URL: [https://kar.kent.ac.uk/14016/](https://kar.kent.ac.uk/14016/).


