A Combination of Clasual and Non Clausal Temporal Logic Program

Shinji Kono

e-mail:kono@csl.sony.co.jp
Sony Computer Science Laboratory Inc.
3-14-13, Higashi-gotanda, Shinagawa-ku, Tokyo 141, Japan

July 12, 1993

We have developped Tokio interpreter[5] for first order Inteval Temporal Logic[11] and an automatic theorem prover [6, 7] for Propositional Interval Temporal Logic. The verifier features deterministic tableau expansion and binary decision tree representation of subterms. Combining these, we can avoid repeated similar clausal form time constraints, and it is possible to execute wider range of specification without time-backtracking.

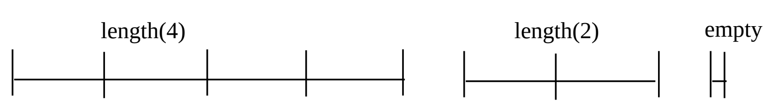
1 Interval Temporal Logic

Interval Temporal Logic[11] (hereafter referred as ITL) uses a sequencing modal operator as its basis. In this logic, it is very easy to express control structures in conventional programming languages, (such as ';', `while` statement). From this point of view a process algebra such as CCS[9] or CIRCAL[8] is also good for control structures, but it does not support negation and declarative expressions (like sometime or always) which are common to Temporal Logic.

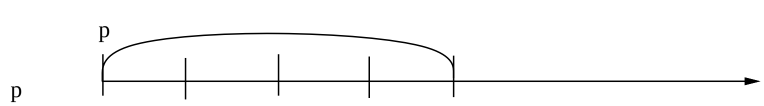
In this paper, we show an implementation of interval temporal logic theorem prover. This method is a tableau driven method[12, 14] and a practical implementation of [5, 7]. It also generates a deterministic state diagram as a verification result.

We have developped first order Interval Temporal Logic interpreter using a kind of clausal form. It is easy to combine the result of verification and the interpreter, since the generated state diagram can be easily translated into clausal form.

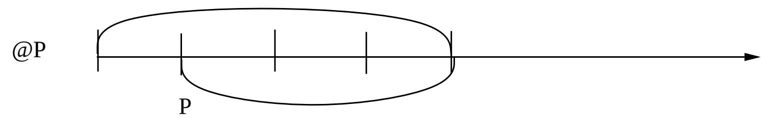
First we show informal visual representation of basic operators in ITL. An interval is a finite line which has number of clock ticks. An operator *empty* is true on the length 0 interval.



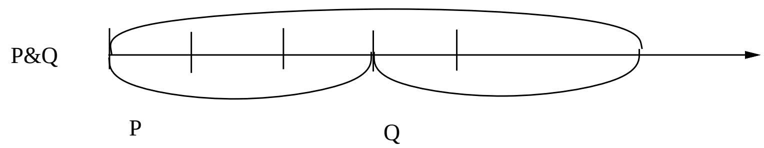
A local variable p means p occurs at the beginning of the interval.



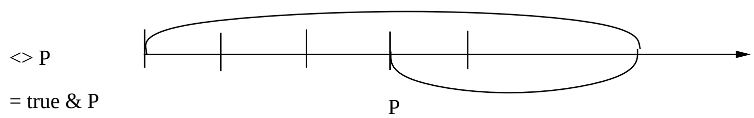
The *nextoperator* $@P$ means P becomes true after one clock cycle. Thus, in ITL $@P$'s interval must be one clock cycle longer than P's and $@P$ is false on the empty interval. We call this *strongnext*. We write *weaknext* $\bigcirc P$ as $@P \vee empty$. P can be any temporal logic formula.

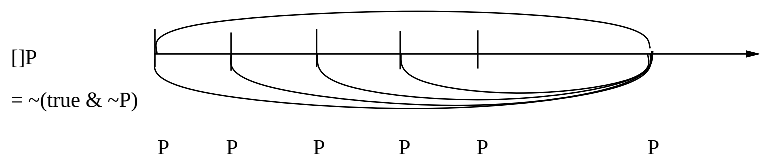


We introduce the chop operator '&' which combines two intervals. $P\&Q$ roughly means "do P then Q".

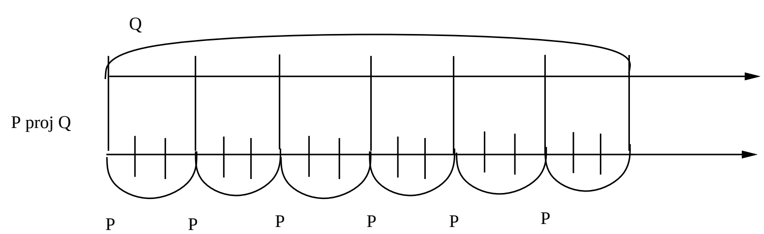


Using the chop operator we can express sometime $\Diamond$ and always $\Box$.





A projection operator creates coarse grain time using a repeated interval. $P\ proj\ \ Q$ means Q is true on a coarse grain time interval. In this interval clock ticks are defined by the repetition of P.



We shall use the following abbreviations,

$$
\begin{aligned}
P \vee Q &\equiv (\neg P) \Rightarrow Q \\
P \wedge Q &\equiv \neg(P \Rightarrow \neg Q) \\
P \Leftrightarrow Q &\equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P) \\
more &\equiv \neg empty \\
+P &\equiv P\&(P \vee empty)\&...\&(P \vee empty) \\
\Diamond P &\equiv T\&P \\
\Box P &\equiv \neg\Diamond\neg P \\
\bigcirc P &\equiv @P \vee empty \\
skip &\equiv @empty \\
length(n) &\equiv \underbrace{@@...@}_{n} empty
\end{aligned}
$$

$$
\begin{aligned}
less(n) &\equiv \underbrace{\bigcirc\bigcirc \ldots \bigcirc}_{n} F \\
\forall P\ f(P) &\equiv \neg\exists P \neg f(P) \\
P \&\& Q &\equiv (P \wedge \neg empty) \& Q \\
*\ P &\equiv (P\ proj\ \ T) \vee (empty \wedge P)\ \text{(closure)} \\
fin(P) &\equiv empty \Rightarrow P \\
halt(P) &\equiv empty \Leftrightarrow P
\end{aligned}
$$

$+P$ is a closure of chop. The *chop standard form* is a formula which all these abbreviations have been removed. Chop standard form may include variables and conjunction, disjunction, negation, chop, projection and existential quantifier operations.

For example, we can make a simple theorem, $\Diamond empty$, since we use finite interval (every interval must include an termination point). Hence, its dual $\Box more$ is unsatisfiable since we cannot extend the interval indefinitely. Later we prove that

$$(\Box\Diamond P) \Leftrightarrow (\Diamond\Box P) \Leftrightarrow fin(P),$$

from which we deduce ITL cannot express fairness. As indicated in [13], the decision procedure is simple for finite intervals.

1.1 Specification in Interval Temporal Logic

In ITL, it is easy to express sequential execution and time out.

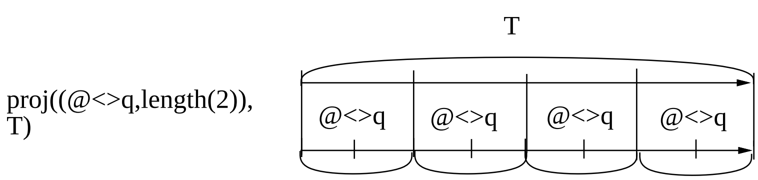
$$((less(5) \wedge \Diamond p \wedge \Diamond q) \vee (length(6) \& s)) \& \Box r$$

This means that p and q have to be done in 5 clock cycles, and after that r stays true until the end of the interval. Otherwise s is happen before r.

Using $proj$, the repeated event and time sharing task are easily described as in [3]. The expression

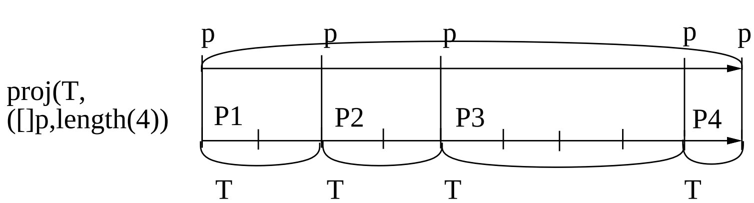
$$(length(2) \wedge \Diamond p)\ proj\ \ T$$

represents a process in which p happens every 2 clock cycles (its timing is not specified).



Conversely some preemptable task p which takes 10 ticks can be represented as follows

$$T\ proj\ \ (\text{length}(4) \wedge \Box p)$$



Of course, we can add a time limit easily. For example, if task p has to be done before q will happen:

$$((T\ proj\ \ (length(4) \wedge \Box p)) \wedge keep(\neg q)) \& q.$$

For a more complex example, if we have 2 periodical tasks (intervals are 3 clocks and 5 clocks) and one time sharing task with dead line with $length(15)$, which shares one resource. (Fig.1).

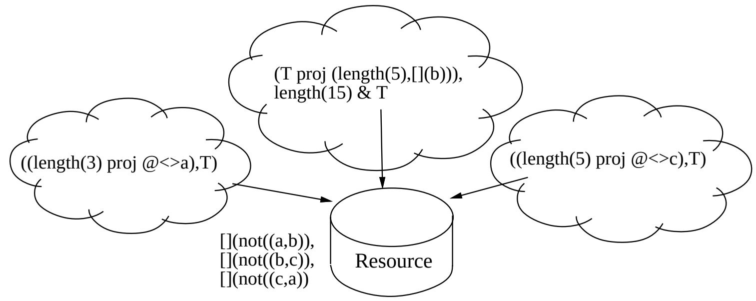


Figure 1: Real-time Task Combination

$$
\begin{array}{l}
((T\ proj\ (length(5) \wedge \Box(c))) \wedge length(15))\&T)\wedge \\
(((length(3)\ proj\ \ @\Diamond(a)) \wedge T)\&less(3))\wedge \\
(((length(5)\ proj\ \ @\Diamond(b)) \wedge T)\&less(5))\wedge \\
\Box(\neg((a \wedge b)))\wedge \\
\Box(\neg((b \wedge c)))\wedge \\
\Box(\neg((c \wedge a)))
\end{array}
$$

2 Verification Methods

To verify a temporal logic, several methods are known such as the Tableau Method [14], Finite Automaton Generation [2] and Model Checking[10, 4].

Local ITL is also known to be decidable [11] using conversion to Quantified Linear Time Temporal Logic. However, this conversion generates one extra variable for each & operator which makes verification space/time hard. It also introduces infinite the interval and fairness unnecessarily. Hereafter we restrict ourselves in Local ITL. There is a model checker for Branching Temporal Logic / Computation Tree Logic [10] which has polynomial order complexity. However this is not a complete verifier.

3 Deterministic Tableau Expansion

In ITL, a temporal logic formula P can be separated into two parts: the current clock period and the future clock period. This separation can be represented by a disjunctive normal form with the *empty* and the @ (strong next) operators.

$$\vdash P \Leftrightarrow (empty \wedge P_e) \vee \bigvee_i P_i \wedge @Px_i$$

A formula P is true on an empty interval if P_e is true. In the case of a non-empty interval, the required condition Px_i at the next clock period depends on the current state condition P_i. P_e and P_i must not contain temporal logic operator. We call this separated form the $@-normalform$. Each P and Px_i represents a possible world, and which are connected by a possible clock transition. To make all possible world, this transformation has to be applied to the generated formula Px_i repeatedly. Termination of this procedure will be discussed in later section.

For exapmle, $@-normalform$ for $p\&q\&r$ is

$$\begin{aligned}\vdash p\&q\&r &\Leftrightarrow (empty \wedge r \wedge q \wedge p)\\ &\vee (r \wedge q \wedge p \wedge @T)\\ &\vee (\neg(r) \wedge q \wedge p \wedge @(T\&r \vee T\&q\&r))\\ &\vee (\neg(q) \wedge p \wedge @(T\&q\&r)).\end{aligned}$$

This $@-normalform$ represents a non-deterministic state transition shown in Fig.2.

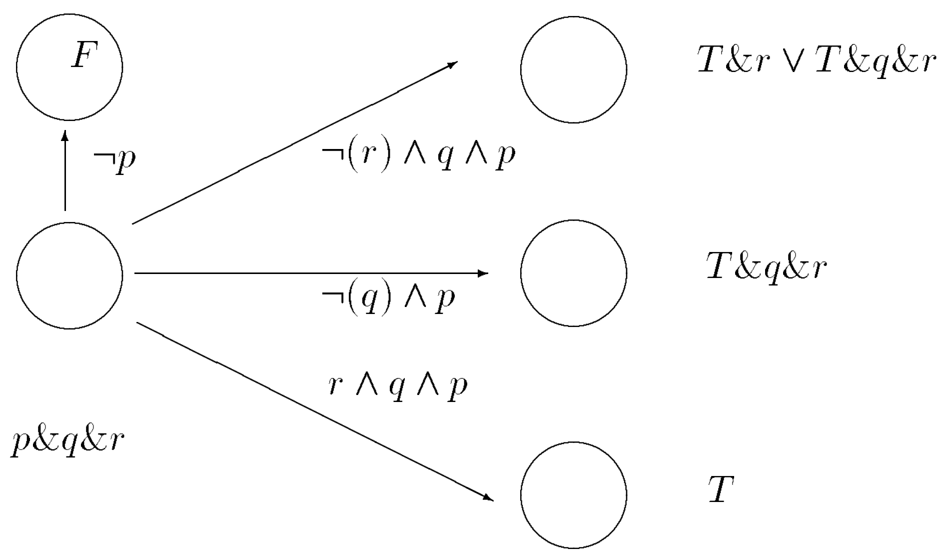


Figure 2: State Transiton for Chop Operator

This separation is performed recursively on temporal logic operators in the formula. For example, if we have two @-normal forms for P and Q then,

$$\begin{aligned}P &= (empty \wedge P_e) \vee \bigvee_i P_i \wedge @Px_i\\ Q &= (empty \wedge Q_e) \vee \bigvee_i Q_i \wedge @Qx_i\end{aligned}$$

The @-normal form for $P\&Q$ will be,

$$P\&Q = (empty \wedge P_e\&Q) \vee \bigvee_i P_i \wedge @Px_i\&Q.$$

The expansion is easy because we use non-deterministic state transition, but there is a problem. Since we use @-normal form (which is a kind of disjunctive normal form) negation becomes expensive. If P contains n disjunction then n-times normalization is necessary to achieve @-normal form. This corresponds the fact that this transformation generates non-deterministic transition.

However, if the conditions P_e, P_i do not overlap each other (i.e. if the transition conditions P_e, P_i are deterministic) negation becomes very easy,

$$\vdash \neg P \Leftrightarrow (empty \wedge \neg P_e) \vee \bigvee_i P_i \wedge @\neg Px_i. \text{ (if } P_e, P_i \text{ do not overlap each other)}$$

We call @-normal form deterministic if the conditions P_e and P_i do not overlap. Fortunately, it is possible to keep deterministic @-normal form in every tableau expansion of an ITL operator.

Since P_i and P_e contain no temporal logic formulae then non-overlapped conditions can be represented as a binary decision tree, in which leaves are ITL formulae. If the condition contains n variables then each node has a maximum of 2^n leaves. We do not need to simplify P_i, P_e part, since the expansion is unique. In fact, for a binary decision tree, P_i, P_e is represented by a path in the tree (i.e a set of variables and *empty* or its negation). If we need two variables a, b for P_e, the possible paths are: $[empty, +a, +b], [empty, +a, -b], [empty, -a, +b], [empty, -a, -b]$. Then We write @-form for P like this:

$$\begin{array}{lllll} P: & [empty, +a, +b] & \rightarrow & P_{e0} \\ & [empty, +a, -b] & \rightarrow & P_{e1} \\ & [empty, -a, +b] & \rightarrow & P_{e2} \\ & [empty, -a, -b] & \rightarrow & P_{e3} \\ & [more, +a, +b] & \rightarrow & P_{x0} \\ & [more, +a, -b] & \rightarrow & P_{x1} \\ & [more, -a, +b] & \rightarrow & P_{x2} \\ & [more, -a, -b] & \rightarrow & P_{x3} \end{array}$$

$\rightarrow$ means a state transition here. P_{ei} are T or F because it contains no temporal logic operator or variables. P_{xi} are temporal logic formulae, which label possible worlds as states. In this way, the tableau expansion can generate a deterministic automaton. To check the finiteness of the automaton, another normal form technique is necessary for the leaves (which will be discussed in the later section).

For fixed P_i, P_e, the deterministic tableau expansion rules can be described as a boolean operation on the leaves. Here we assume P's leaf for an P_i condition is $more(P)$ and P's leaf for a P_e condition is $empty(P)$. If we meet a local variable p, a node is added to the binary decision tree, that is, P_i is changed into two leaves $P_i \wedge p$ and $P_i \wedge \neg p$. Since $empty(P)$ contains no ITL operator, no variable and no connectives, $empty(P)$ is T or F.

$$\begin{array}{llll} T & & & \\ & empty(T) & = & T \\ & more(T) & = & @T \\ P \wedge Q & & & \\ & empty(P \wedge Q) & = & empty(P) \wedge empty(Q) \\ & more(P \wedge Q) & = & more(P) \wedge more(Q) \\ P \vee Q & & & \\ & empty(P \vee Q) & = & empty(P) \vee empty(Q) \\ & more(P \vee Q) & = & more(P) \vee more(Q) \\ \neg P & & & \\ & empty(\neg P) & = & \neg empty(P) \\ & more(\neg P) & = & \neg more(P) \\ @P & & & \\ & empty(@P) & = & F \\ & more(@P) & = & @P \\ P\&Q & & & \\ & empty(P\&Q) & = & empty(P) \wedge empty(Q) \\ & more(P\&Q) & = & (empty(P) \wedge more(Q)) \vee (more(P)\&Q) \end{array}$$

$$
\begin{array}{lll}
\exists y Q & & y \text{ is removed from leaf conditions} \\
\quad empty(\exists y Q) & = & (empty(y \wedge Q) \vee empty(\neg y \wedge Q)) \\
\quad more(\exists y Q) & = & \exists y((more(y \wedge Q) \vee more(\neg y \wedge Q))) \\
*(P) & & \\
\quad empty(*(P)) & = & empty(P) \\
\quad more(*(P)) & = & more(P) \& *(P) \\
P \; proj \;\; Q & & \\
\quad empty(P \; proj \; Q) & = & empty(Q) \\
\quad more(P \; proj \; Q) & = & more(P) \& (P \; proj \; more(Q))
\end{array}
$$

These transformation rules are part of the complete axiom system in ITL. To see the soundness of these transformations, we have to look at the model definition of the temporal logic operator.

For the chop rule, we have

$$
\begin{aligned}
M_{ij}(P\&Q) \;\; = \;\; & T \text{ when } i \leq \exists k \leq j, \\
& M_{ik}(P) = T, M_{kj}(Q) = T \\
& F \text{ otherwise.}
\end{aligned}
$$

First we note that $M_{ij}((P \vee Q)\&R) = T$ if $M_{ij}(P\&R) = T$ or $M_{ij}(Q\&R) = T$. Since leaves in a binary tree are all connected by disjunctions, a proof on a leaf is sufficient.

In the case of empty, $i = j$ in $M_{ij}(P\&Q) = T$, then i can be used as a k, and $M_{ii}(P) = T$ and $M_{ii}(Q) = T$. This is equivalent to the $M_{ii}(P) \wedge M_{ii}(Q)$. That is $empty(P\&Q) = empty(P) \wedge empty(Q)$. If $j > i$ then $k = i$ or $k > i$. In case of $k = i$, $M_{ik}(P) = T$ and $M_{kj}(Q) = T$ are necessary, so that $empty(P) = T$ and $more(Q)$. Otherwise $k > i$, requires $more(P)\&Q$. QED.

Since all the mapping function has distribution rule for disjunciton, other rules can be proved in the same way. This is because ITL's chop operator has the same property as existential quantifier.

3.1 Expansion Example

The tableau expansion of $p\&q$ generates a tree with 6 leaves. For the empty condition we can replace & with $\wedge$. Then we have

$$
\begin{array}{llll}
empty \wedge (p\&q): & [empty, +q, +p] & \rightarrow & T \\
& [empty, -q, +p] & \rightarrow & F \\
& [empty, -p] & \rightarrow & F.
\end{array}
$$

For the non-empty condition,

$$
\begin{array}{llll}
more \wedge (p\&q): & [more, +q, +p] & \rightarrow & T \\
& [more, -q, +p] & \rightarrow & (T\&q) \\
& [more, -p] & \rightarrow & F.
\end{array}
$$

The first line comes from $empty(P \wedge Q) \vee (more(P)\&Q)$ and $empty(P \wedge Q) = T$. The second line comes from $empty(P \wedge Q) = F$ and $more(P) = T$.

3.2 Binary Subterm Tree

During possible world generation, various kind of ITL formulae are generated. Unlike LTTL or ETL [14], generated formulae contain more complex terms than the original subterm. It is not easy to see the finiteness of generated formulae.

To overcome this situation, we introduce a binary subterm tree. This subterm tree contains typed nodes:

- A triple $?(P, Q, R)$ is a binary decision node, in which if variable P is T then Q else R.
- $\exists x Q$, where x is a free variable.
- a numbered node for a unary temporal logic operator $O(P)$. (ex. $@, *$)
- a numbered node for a binary temporal logic operator $O(P, Q)$. (ex. &, *proj*)

Translation from ITL formula to binary subterm tree is done bottom-up. For example, $\Diamond\Box p$ is expanded into a chop standard form: $T\&\neg(T\&\neg p)$. First $\neg p$ is translated into,

$$?(p, F, T).$$

Then we need a numbered node s_1 for the chop operator, such that,

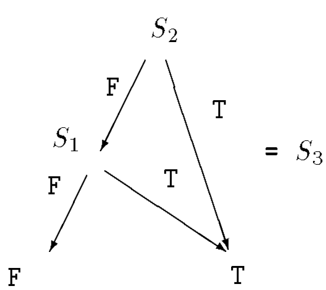
$$s_1 = T\&?(p, F, T).$$

Then the original formula is transformed into a numbered node, such that,

$$s_2 = T\&?(s_1, F, T).$$

After tableau expansion of this formula, we have a complex formula, $(\neg(T\&\neg p)) \vee (T\&\neg(T\&\neg p))$. But the result of the transformation is simple (Fig. 3),

$$s_3 = ?(s_2, T, ?(s_1, F, T)).$$



$$\underbrace{\neg(\underbrace{(T\&not(p))}_{s_1}) \vee \underbrace{T\&\neg(\underbrace{T\&\neg(p)}_{s_1})}_{s_2}}_{s_3}$$

Figure 3: Binary Subterm Tree

In this way, we can store generated formulae compactly in a binary subterm tree. As with the binary decision diagram, if we fix the ordering of nodes from top to bottom, the form of a node becomes unique to the logical connectives such as negation, conjunction or disjunction.

3.3 Termination of tableau expansion

If binary subterm trees contain finite numbered nodes, a set of the binary subterm trees must be finite, During the tableau expansion, we generates a formula which contains temporal logic operators. If this generated formula contains a new form of binary subterm tree in the argument of the operator then it may require a new node.

The expansion rules for logical connectives and the operator do not increase numbered nodes. Other rules generate only a fixed amount of temporal logic operators. Although we do allow recursion, we ensure that the depth of the temporal logic operator is monotonically decreased in an argument. For example, in projection the former part decrease the depth monotonically but the latter part increases by a single & operator. Subsequently there are no infinitely applied rules, and only a finite number of new temporal logic operators are generated.

Here we prove this for the *proj* operator. Others can be proved in the same way.

Suppose $more(P), more(Q)$ generate finite variant. In the tableau expansion of *proj* ,

$$more(P\ proj\ Q) = more(P)\&(P\ proj\ more(Q)),$$

it generates a new node for the chop operator and the projection operator. The former part of the chop $more(P)$ can vary according to the variant. The latter part of the chop can vary according to the variant of $more(Q)$. So the number of generated formulae is less than the number of products of variants for $more(P)$ and $more(Q)$. QED.

Since the tableau expansion generates a finite number of binary subterm tree nodes, it generates only a finite binary subterm tree. When we expand all binary subterm trees, the expansion completes.

3.4 Verification Example

First try to prove $fin(p) \Leftrightarrow \Box(\Diamond\ p)$. The chop normal form is rather complex:

$$(\neg(T\&(empty \wedge p)) \vee \neg(T\&\neg(T\&p))) \wedge (T\&(empty \wedge p) \vee T\&\neg(T\&p))$$

We number this state as S_1. It looks complex but it has a simple binary subterm tree form. It has only three &, so we have three binary subterm nodes:

$$\begin{array}{lllll} s_1: & T\&?(p,?(empty,T,F),F)) & = & T\&(empty \wedge p) \\ s_2: & T\&?(p,T,F)) & = & T\&p \\ s_3: & T\&?(s_2,F,T)) & = & T\&\neg(T\&p). \end{array}$$

The original formula has binary subterm tree: $?(s_3,?(s_1,F,T),?(s_1,T,F))$.

It is translated into @-normal form.

$$\begin{array}{llll} S_1: & [empty,+p] & \rightarrow & T \\ & [empty,-p] & \rightarrow & T \\ & [more,+p] & \rightarrow & S_1 \\ & [more,-p] & \rightarrow & S_2 \end{array}$$

S_2 is a newly generated state,

$$(\neg(T\&(empty \wedge p)) \vee \neg((\neg(T\&p) \vee T\&\neg(T\&p)))) \wedge (T\&(empty \wedge p) \vee \neg(T\&p) \vee T\&\neg(T\&p)).$$

Its subterm tree form is $?(s_3, ?(s_1, F, T), ?(s_2, ?(s_1, T, F), ?(s_1, F, T)))$, and it is expanded into:

$$\begin{array}{llcl} S_2: & [empty, +p] & \rightarrow & T \\ & [empty, -p] & \rightarrow & T \\ & [more, +p] & \rightarrow & S_1 \\ & [more, -p] & \rightarrow & S_2 \end{array}$$

This is exactly the same as S_1's transition function. There are no newly generated formula in this case, so we can finish the tableau expansion procedure. It is easy to check every empty leaf has T, so the original formula is valid.

4 Generated Model and Counter Example Generation

After tableau expansion, we have a deterministic finite automaton. Each state in an automaton is labelled by a unique binary subterm tree. The transition condition of the automaton is a list of +variable and -variable. The initial state is labelled by the original formula, and the empty transition (i.e. transition in which condition contains the empty operator) generates T or F.

This automaton returns T or F for finite series of events represented by local interval temporal logic variables. That is, it characterize the original temporal logic formula. Since we are handling finite sequences, the termination condition of an automaton is a part of the assumption.

The output automaton state indicates which subterm is true or false; it can be used as a specification tester in hardware implementation.

If for all transitions which contain empty results T then it must accept all possible sequences of truth value assignments for the variables. That is, the original formula is valid. If there are no T nodes in the empty transition then the original formula is unsatisfiable.

Once we have an F node in the empty transition, it is easy to generate a counter example. The problem is how to discover a path from the initial state to the F node. The shortest path is easily found, if we reverse all links and mark the states in the automaton with a number in generated order.

First we check the least numbered F node, then pick up the least numbered node from the reverse links from the F node. We repeat tracing the least numbered node in the reverse links. Eventually we will reach the initial node which has the least numbered node in the root of reverse links. The generated shortest path should be acyclic. The traced path represent the shortest counter execution. If we start from a T node, we will have the shortest sample execution. Unfortunately, the shortest examples need not necessarily be useful.

4.1 Execution Examples

Consider next formula:

$$(p\&\&p\&\&p\&\&p\&\&p\&\&\neg p\&\&p) \rightarrow \Box(\Diamond p).$$

Using our verification program, it generates 13 states, 13 subterms, 42 state transition. It takes 13.7 sec on T2200SX, 386sx IBMPC compatible Laptop. There is a counter example because p is a local variable and there are no constraints on the end of an interval on the assumption in the formula. It is possible to make p false at the end of an interval which violates $\Box\Diamond p$, that is $fin(p)$.

— ?- diag.
counter example:
0:+p 2
1:+p 3
2:+p 5
3:+p 7
4:+p 9
5:-p 11
6:+p 12
7:-p F

Figure 4: Example of Counter Example Generation

The next one is more complex.

$$
\begin{array}{l}
(((T\ proj\ \ (length(5) \wedge \Box(dc))) \wedge length(15))\&T)\wedge \\
((length(3)\ proj\ \ @\Diamond(ac)) \wedge T)\wedge \\
((length(5)\ proj\ \ @\Diamond(bc)) \wedge T)\wedge \\
((length(5)\ proj\ \ @\Diamond(cc)) \wedge T)\wedge \\
\Box(((ac \wedge \neg(bc) \wedge \neg(cc) \wedge \neg(dc))\vee \\
(\neg(ac) \wedge bc \wedge \neg(cc) \wedge \neg(dc))\vee \\
(\neg(ac) \wedge \neg(bc) \wedge cc \wedge \neg(dc))\vee \\
(\neg(ac) \wedge \neg(bc) \wedge \neg(cc) \wedge dc)\vee \\
(\neg(ac) \wedge \neg(bc) \wedge \neg(cc) \wedge \neg(dc))))
\end{array}
$$

The first line is a time sharing task which has a deadline $length(15)$ and requires 5 clock cycles to be done. Between the second and the fourth lines are 3 periodical tasks. The reminder consists of shared resource conditions, that is, ac, bc, cc and dc cannot happen together. If it has no possible execution, these real-time tasks are not schedulable.

Using the verifier, it generates 242 states, 115 subterms and 799 state transition. It takes 336.591 sec on 386sx 20Mhz, and it finds there are no possible executions. If we change the load on time sharing task from 5 clock cycle to 4 clock cycle it finds a possible execution. This time it requires 392.28 sec and it generates 303 states, 107 subterms and 982 state transition. A generated possible execution is shown in Fig. 5.

The implementation includes X window Interface as shown in Fig. 6.

5 Combination of Clasusal Form Program and Constraints

First order logic version of Interval Temporal Logic interpreter is called Tokio. The relationship between Tokio and first order ITL is just like the one between Prolog and first order predicate logic. If we do not use temporal operators, the execution of Tokio is just the same as that of Prolog.

The syntax of Dec-10 Prolog is chosen here [1].

$$ap([], X, X). \tag{1}$$

$$ap([H|X], Y, [H|Z]) \quad :- \quad ap(X, Y, Z). \tag{2}$$

This is a Prolog append program. [X|Y] means a cons pair and [] means nil. A form *ap(...)* is called a predicate. Variables in Tokio are words beginning with capital letters or _.

execution:
0:-ac-bc-cc+dc 2
1:-ac-bc-cc+dc 3
2:-ac-bc-cc+dc 8
3:+ac-bc-cc-dc 13
4:-ac+bc-cc-dc 16
5:-ac-bc+cc-dc 19
6:+ac-bc-cc-dc 20
7:-ac-bc-cc+dc 21
8:+ac-bc-cc-dc 27
9:-ac+bc-cc-dc 31
10:-ac-bc+cc-dc 33
11:+ac-bc-cc-dc 35
12:-ac+bc-cc-dc 50
13:+ac-bc-cc-dc 52
14:-ac-bc+cc-dc 54
15:-ac-bc-cc+dc 0

Figure 5: A Possible Execution

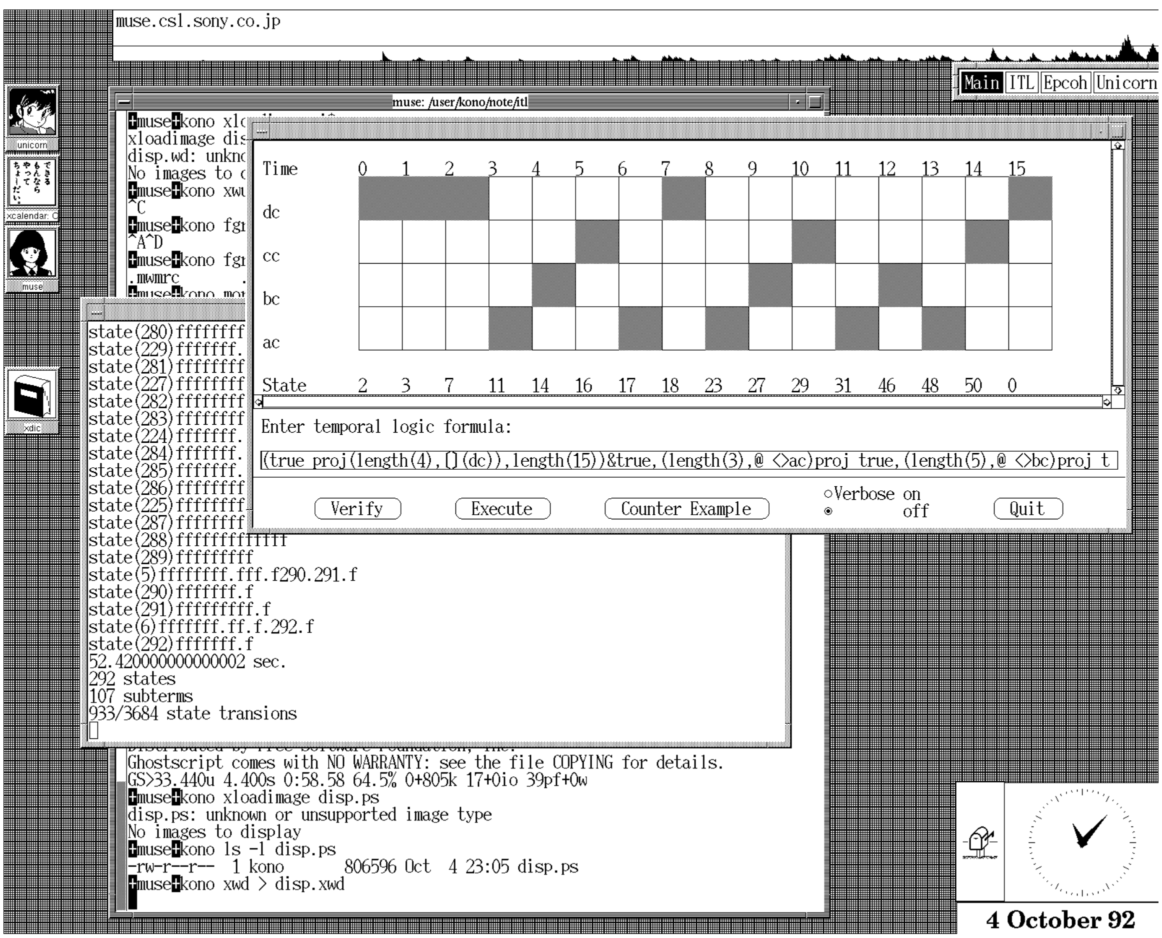


Figure 6: X window display

A Tokio program: $p1, p2, ..., pn$ is true in the interval int is described as:

$$int :- p1, p2, ..., pn.$$

That is, the interval name is used as a predicate name, and p1, p2, ..., pn are used as bodies. The Horn clause in Tokio is extended with temporal operators, such as

$$t1 :- \Box write(0), length(3) \&\& \Box write(1), length(5).$$

This writes four '0' and six '1' for each clock period.

6 Combination Example

Let's consider a toy GUI (Graphical User Interface) (Fig. 7). There are two buttons named `start`

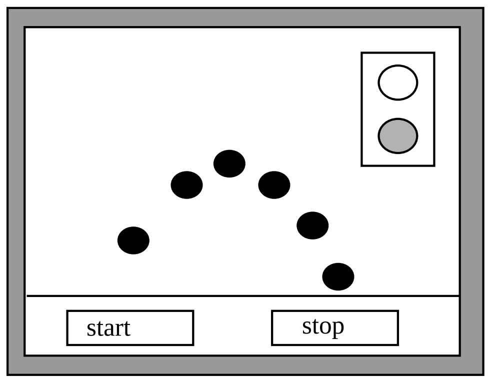


Figure 7: A toy GUI

and `stop` and one signal. When `start` button is pushed the signal turns green and the GUI starts moving a small ball. When `stop` button is pushed the signal turns red and the ball stops. The specification is expressed in ITL like this:

$$
\begin{array}{l}
+(((stop \wedge keep((red \wedge \neg(start))) \vee start \wedge keep((green \wedge \neg(stop))))))\wedge \\
\Box((red \wedge \neg(green) \vee \neg(red) \wedge green))\wedge \\
\Box((green \rightarrow move))\wedge \\
\Box((red \rightarrow \neg(move)))halt(quit)
\end{array}
$$

This is translated into state diagram using our verifier. It takes 2.97 sec in 386SX 20Mhz IBMPC and generates 3 states, 10 subterms, 13/90 state transitions. The specification contains many variables, so state transitions are large, but actual states are small.

The resulted state transitons are automatically translated into Tokio program. Generation of clausal form is straight forward. Here we use * for static variable in Tokio interpreter. The events are classified for input events and output events. Input events are translated into equation and output events become assignments.

```
s1 :- empty,*stop= 1,*quit= 1,*green:= 0,*move:= 0,*red:= 1,empty.
s1 :- empty,*stop= 1,*quit= 1,*green:= 1,*move:= 1,*red:= 0,empty.
s1 :- empty,*stop= 0,*start= 1,*quit= 1,*green:= 0,*move:= 0,*red:= 1,empty.
s1 :- empty,*stop= 0,*start= 1,*quit= 1,*green:= 1,*move:= 1,*red:= 0,empty.
```

```
s1 :- more,*stop= 1,*start= 0,*quit= 0,*green:= 0,*move:= 0,*red:= 1,@s2.
s1 :- more,*stop= 0,*start= 1,*quit= 0,*green:= 1,*move:= 1,*red:= 0,@s3.
s2 :- empty,*quit= 1,*green:= 0,*move:= 0,*red:= 1,empty.
s2 :- empty,*quit= 1,*green:= 1,*move:= 1,*red:= 0,empty.
s2 :- more,*stop= 1,*start= 0,*quit= 0,*green:= 0,*move:= 0,*red:= 1,@s2.
s2 :- more,*stop= 0,*start= 0,*quit= 0,*green:= 0,*move:= 0,*red:= 1,@s2.
s2 :- more,*stop= 0,*start= 1,*quit= 0,*green:= 1,*move:= 1,*red:= 0,@s3.
s3 :- empty,*quit= 1,*green:= 0,*move:= 0,*red:= 1,empty.
s3 :- empty,*quit= 1,*green:= 1,*move:= 1,*red:= 0,empty.
s3 :- more,*stop= 0,*start= 1,*quit= 0,*green:= 1,*move:= 1,*red:= 0,@s3.
s3 :- more,*stop= 0,*start= 0,*quit= 0,*green:= 1,*move:= 1,*red:= 0,@s3.
s3 :- more,*stop= 1,*start= 0,*quit= 0,*green:= 0,*move:= 0,*red:= 1,@s2.
```

We can test the generated implementation for a set of input events using a Tokio program. Next program generates a `start` event and a `stop` event.

```
test :-
        static([green, red, move, stop, start]),
        *green:=0, *red:=1, *move:=0, *stop := 1, *start :=0 , *quit := 0
        && ((
            length(3), *stop := 0, *start :=1 &&
            length(3), *stop := 0, *start :=0 &&
            *stop := 1, *start :=0, @(*quit := 1)
            ),
            s1,
            []((G= *green, R= *red, S= *start, P= *stop, M = *move,
                write((green,G,start,S,red,R,stop,P,move,M))))).
```

Here is a possible execution result.

```
?- tokio test.

t0:
t1:green,0,start,0,red,1,stop,1,move,0
t2:green,0,start,1,red,1,stop,0,move,0
t3:green,1,start,1,red,0,stop,0,move,1
t4:green,1,start,1,red,0,stop,0,move,1
t5:green,1,start,0,red,0,stop,0,move,1
t6:green,1,start,0,red,0,stop,0,move,1
t7:green,1,start,0,red,0,stop,0,move,1
yes
| ?-
```

The example looks small but it becomes large quickly if we add more time constraints. But our verifier is robust for large rules and small number of variables. Since GUI prefer small number of buttons to control complex control such as double click, it seems that this method fits for GUI application generator. Especially simple adding of temporal constraints are far easier than modications of the state diagram, which is buggy part of GUI.

A part of actual GUI code is shown below. This GUI uses InterViews package of SICStus Prolog.

```
toy :-  static([green, red, move, stop, start,quit]),
        *move:=1,*quit:=0,*stop:=0,*start:=1,*red:=0,*green:=1,
        bounce_init(W,R,G),@toy1(W,R,G).
toy1(W,R,G) :-
        [](event),                                      % input
        s1,                                             % automaton
        []((button_red(R),button_green(G),bounce(W))).  % output

event :- nextevent(E),E=E1,event_select(E1).
event_select(noevent) :- true.
event_select(button(_,start)) :- *start := 1, *stop  := 0.
event_select(button(_,stop)) :-  *stop  := 1, *start := 0.
event_select(button(_,quit)) :-  *quit := 1.

button_red(Out) :-
        *red =0, Out => out("").
button_red(Out) :-
        *red =1, Out => out("Red").
```

`event/0` generates input variables such as `*start`. `s1/0` handles state transition. `button_red/1` is an object which handles necessary display procedures. Unlike other GUI package, event handling is seprated from GUI objects such as button or bounce procedure. Since GUI is inherentry parallel, a combination of temporal logic and object oriented graphic package seems useful.

7 Future Works

To make things faster, since the subterm vector is used in the BDD implementation, it is possible to use a set of BDDs rather than a single huge BDD. A modication of existing state diagram is important, because synthesized state transitions can be very large.

References

[1] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[2] Gjalt G. de Jong. An Automata Theroretic Approach to Temporal Logic. In *Computer Aided Verification*. Springer-Verlag, July 1991. 3rd International Workshop, CAV'91.

[3] Roger Hale. Temporal logic programming, 1988.

[4] Hiromi Hirahashi. Design Verification of Sequential Machines Based on a Model Checking Algorithm of ε -free Regular Temporal Logic. Technical Report CMU-CS-88-195, Department of Computer Science, Carnegie-Mellon University, September 1988.

[5] S. Kono, T. Aoyagi, M. Fujita, and H. Tanaka. Verification of Temporal Logic Programming Language Tokio. In *Logic Programming Conference '86*, 1986. (in Japanese).

[6] Shinji Kono. Automatic Verification of Interval Temporal Logic. Technical Report SCSL-TM-92-007, Sony Computer Science Laboratory Inc., October 1992. 第 9 回「記号論理学と情報科学」研究集会.

[7] Shinji Kono. Automatic verification of interval temporal logic. In *8th British Colloquium For Theoretical Computer Science*, March 1992.

[8] G.J. Milne. CIRCAL: A Calculus for Circuit Description. *Integration*, Vol. 1, No. 2, pp. 121–160, 1983.

[9] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Note in Computer Science*. Springer-Verlag, 1980.

[10] B. Mishra and E.M. Clarke. Automatic and hierarchical verification of asynchronous circuits using temporal logic. Technical Report CMU-CS-83-155, Dept. of Computer Science, Carnegie-Mellon Univ., September 1983.

[11] B.C. Moszkowski. Reasoning about digital circuit. Technical Report No.STAN-CS-83-970, Dept. of C.S. Stanford Univ, July 1983.

[12] Nicholas Rescher and Alasdair Urquhart. *Temporal Logic*. Springer-Verlag, 1971.

[13] Roni Rosner and Amir Pnueli. A choppy logic, 1986.

[14] P. Wolper. Synthesis of communicating processes from temporal logic specifications. Technical Report STAN-CS-82-925, Stanford University, 1982.