Automatic Verification of Interval Temporal Logic

Shinji Kono

The University, Newcastle upon Tyne, U.K.

e-mail:kono@ie.u-ryukyu.ac.jp
Information Engineering, University of the Ryukyus
Nishihara-cyo 1, Okinawa, 903-01, Japan

24 Apr 1994

We present two automatic theorem provers for Propositional Interval Temporal Logic[Mos83]. These implementations include a projection operator and a quantifier in Local Interval Temporal Logic. The first version features deterministic tableau expansion and binary decision tree representation of subterms. It requires exponential computational complexity dependent on the number of variables and subterms, but works faster than a näive automaton generation method. The second version is based on automaton generation and the binary decision diagram method. This is also applicable to the Model Checking[MC83]. The verifier outputs a finite automaton for acceptable states, which can be used as an implementation of logic circuit.

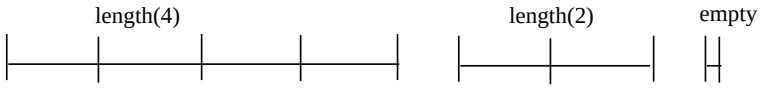
1 Interval Temporal Logic

Interval Temporal Logic[Mos83] (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[Mil80] or CIRCAL[Mil83] 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.

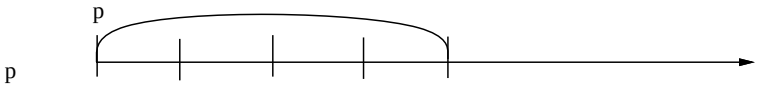
In this paper, we show an implementation of interval temporal logic theorem prover. This method is a tableau driven method[RU71, Wol82] and a practical implementation of

[KAFT86, Kon92]. It uses deterministic tableau method and binary subterm tree, which is suitable for binary decision diagram[Bry86] based implementation.

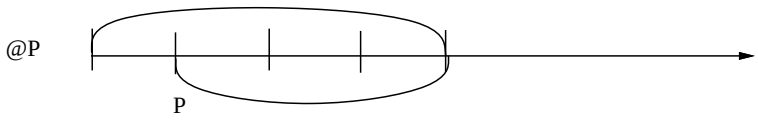
First we show informal visual representation of basic operators in ITL. An interval is a finite line which has number of clock ticks. An operator *empty* is true on the length 0 interval.



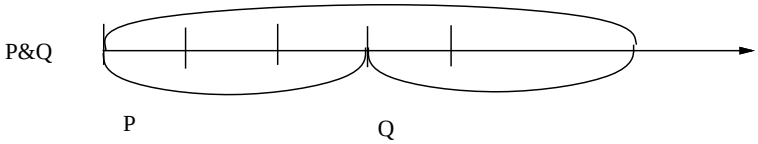
A local variable p means p occurs at the beginning of the interval.



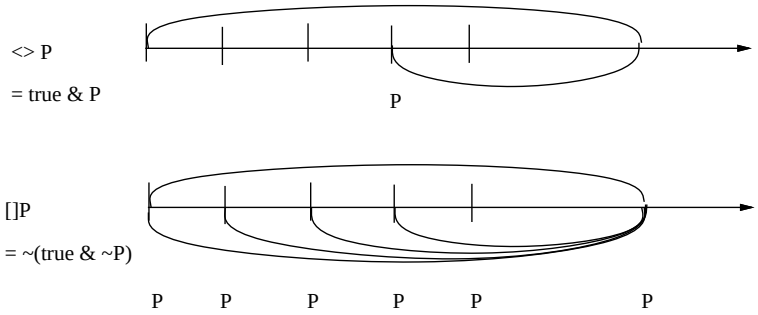
The *nextoperator* @P means P becomes true after one clock cycle. Thus, in ITL @P's interval must be one clock cycle longer than P's and @P is false on the empty interval. We call this *strongnext*. We write *weaknext* $\bigcirc P$ as @$P \vee empty$. P can be any temporal logic formula.



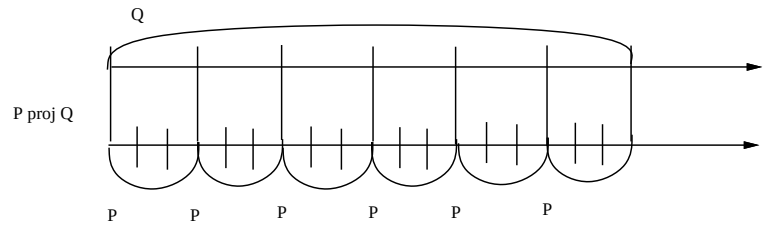
We introduce the chop operator '&' which combines two intervals. $P\&Q$ roughly means "do P then Q".



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.



Now we are ready to define the semantics of ITL in a formal way. An interval is a finite sequence of states. An interval which is a part of some other interval is called a subinterval. In a model of ITL, a series of states beginning from clock 0 to finite future time determines all the states of its subintervals. Such subintervals are determined by two indices which define the start and termination clock periods. So we use a mapping function $M_{ij} : P \rightarrow \{T, F\}$ which has two integer time indices, where $i \leq j$.

Here we use capital letter P, Q, R for temporal logic formulae and small letter p, q, r for propositional variables.

$$
\begin{aligned}
M_{ij}(T) &= T \\
M_{ij}(F) &= F \\
M_{ij}(p) &= \forall j \text{a variable p has a value } T \text{ or } F \\
M_{ij}(P \Rightarrow Q) &= T \text{ when } M_{ij}(P) = F \\
&\quad \text{or } M_{ij}(Q) = T \\
M_{ij}(\exists p, f(p)) &= \text{for a free variable p in f(p), there is a mapping} \\
&\quad \text{function } M' \text{ which contains a variable } p \\
&\quad M'_{ij}(f(p)) = T \\
M_{ij}(P \& Q) &= \begin{cases} T \text{ when } i \leq \exists k \leq j, M_{ik}(P) = T, M_{kj}(Q) = T \\ F \text{ otherwise} \end{cases} \\
M_{ij}(@P) &= \begin{cases} T \text{ when } i+1 \leq j, M_{(i+1)j}(P) = T \\ F \text{ otherwise} \end{cases} \\
M_{ij}(empty) &= T \text{ when } i = j, F \text{ otherwise} \\
M_{ij}(P\, proj Q) &= \text{there is a length } n \text{ ordered sequence } k \text{ of integer such as,} \\
&\quad 0 \leq \exists n \leq j - i, \forall\ r \text{ in } 0..n-1, i \leq k_r \leq j, k_r < k_{r+1} \\
&\quad \text{This sequence is constrained by P,} \\
&\quad M_{k_r k_{r+1}}(P) = T \\
&\quad \text{Q is true on the coarse grain time,} \\
&\quad \forall\ s \text{ in } 0..n \text{ there are some mapping functions } M'_{rs} = M_{k_r k_s} \\
&\quad M'_{0n}(Q) = T \\
M_{ij}(\text{local}(P)) &= = T, \text{if } i \leq \forall k \leq j, M_{ik}(P) = M_{ii}(P)
\end{aligned}
$$

local(P) means P is independent of the termination clock period (i.e. P is dependent

on local time). We use Local ITL since our variables do not depend on the termination time. If the value of a variable is dependent on the termination time, the logic becomes undecidable.

We shall use the following abbreviations,

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

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 [RP86], 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.

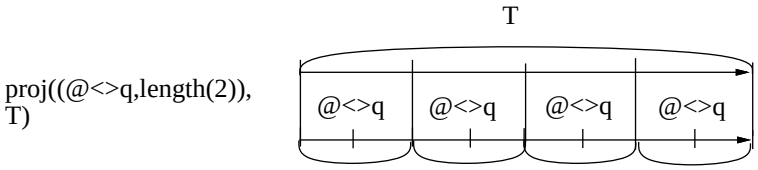
$$((less(5) \wedge \Diamond p \wedge \Diamond q) \vee (length(6) \& s)) \& \Box r$$

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 [Hal88]. The expression

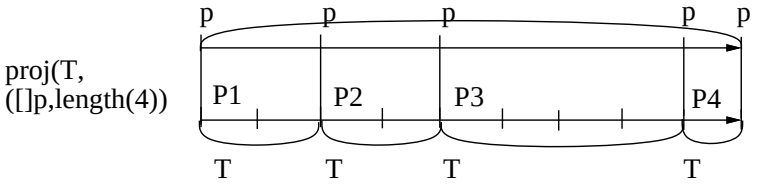
$$(length(2) \wedge \Diamond p)\, proj\ T$$

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).

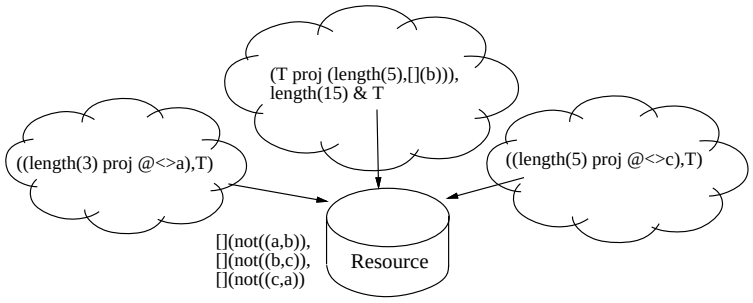


Figure 1: Real-time Task Combination

$$((T\,proj(length(5) \wedge \Box(c))) \wedge length(15))\&T) \wedge$$
$$(((length(3)\,proj\ @\Diamond(a)) \wedge T)\&less(3)) \wedge$$
$$(((length(5)\,proj\ @\Diamond(b)) \wedge T)\&less(5)) \wedge$$
$$\Box(\neg((a \wedge b))) \wedge$$
$$\Box(\neg((b \wedge c))) \wedge$$
$$\Box(\neg((c \wedge a)))$$

2 Verification Methods

To verify a temporal logic, several methods are known such as the Tableau Method [Wol82], Finite Automaton Generation [dJ91] and Model Checking[MC83, Hir88].

Linear Time Temporal Logic (LTTL) is decidable in polynomial order computational complexity using the tableau method. However, quantified LTTL requires more than P-space complexity since the satisfiability problem on quantified boolean formulae is already P-space complete. Finite automaton generation is implemented for LTTL but has exponential complexity.

Local ITL is also known to be decidable [Mos83] 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 [MC83] which has polynomial order complexity. However this is not a complete verifier.

Here we look these methods in more detail.

2.1 Automaton Method

To express a model of temporal logic, a finite automaton (FA) can be used. A FA consists of a set of state $\{S_n\}$ and conditions P_{mn} on $S_m \rightarrow S_n$ transition. $\{S_n\}$ contain T, F and an initial state S_1. Transition conditoins are boolean formula of propositional variables and $empty, more$. We can use deterministic finite automaton (DFA) without loss of expressiveness, which trasition conditions are exclusive each other. To express the exclusive transition condition, a binary tree on propositional variables can be used.

A variable p in temporal logic can be expressed as a simple automaton, such that, in the initial state S_1,

$$P_{0T}: \quad S_1 \rightarrow T \text{ if } p,$$
$$P_{0F}: \quad S_1 \rightarrow F \text{ if } \neg p.$$

In ITL, emptiness of the interval has to be checked, then we have following conditions.

$$
\begin{aligned}
P_{0T}: &\quad S_1 \rightarrow T \text{ if } p \wedge empty, \\
P_{0F}: &\quad S_1 \rightarrow F \text{ if } \neg p \wedge empty. \\
P_{0T}: &\quad S_1 \rightarrow T \text{ if } p \wedge more, \\
P_{0F}: &\quad S_1 \rightarrow F \text{ if } \neg p \wedge more.
\end{aligned}
$$

The transition condition can be represented in a binary decision tree in Fig.(2), which has four leaves. To represent the tree we can use a list of propositional variable and

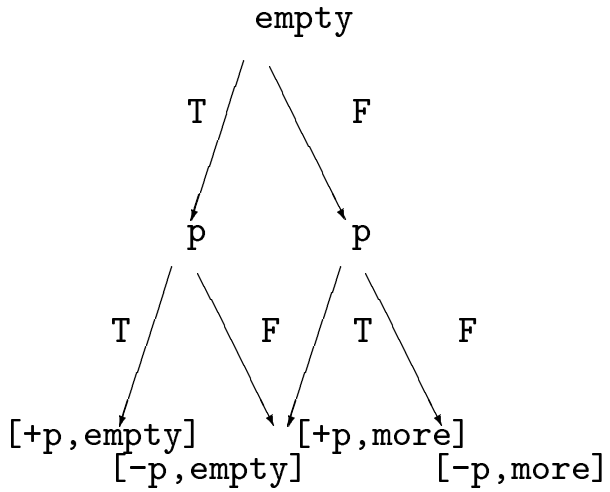


Figure 2: Binary Decision Tree for Transition Condition

empty for a leaf of the tree where $+p$ means p is true and $-p$ means p is false in the leaf.

$$
\begin{aligned}
S_1: &\quad [+p, empty] \rightarrow T \\
&\quad [-p, empty] \rightarrow F \\
&\quad [+p, more] \rightarrow T \\
&\quad [-p, more] \rightarrow F
\end{aligned}
$$

Since & can be considered as a concatenation of automata, the entire automaton for an ITL formula is easily constructed as follows:

chop concatenetaion

conjunction products

disjunction union

negation negation of automaton

In the case of LTTL, since it requires infinite time structure, Büchi automaton or Müller automaton is necessary. But in ITL a finite automaton is enough for finite interval (Fig.3).

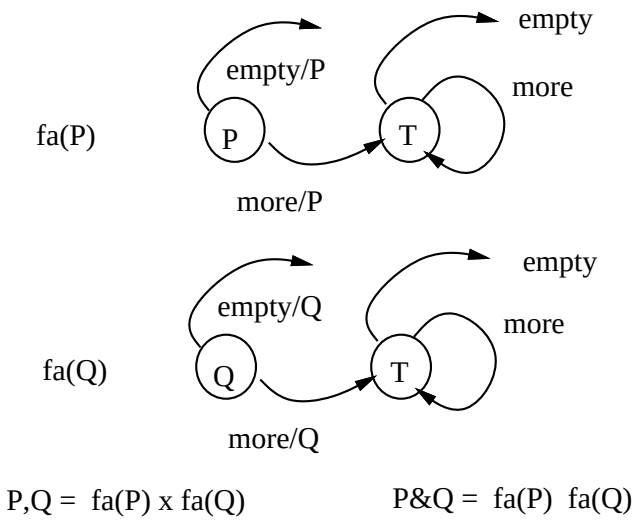


Figure 3: Automaton Method

The greatest disadvantage of the automaton method is the fact that negation of automata requires non-deterministic/deterministic conversion. As shown in [dJ91], we cannot avoid determinisation with combinations of disjunction and negation, which are very common in $P \Rightarrow Q$ type specifications. A known determinisation method is the subset method, which takes exponential computation complexity to the number of the original states.

The automaton method generates an empty automaton for valid formulae, but intermediate finite automata can be very large due to determinisation.

2.2 Tableau Method

Temporal logic operators define relationships between possible worlds. Each possible world is characterized by a temporal logic formula [RU71, Gol87]. If the characterization of possible worlds is finite we can construct all possible world relations and it is possible to check validity (i.e. we have decision procedure). In this sense the tableau method has two parts: the first is possible world generation and the second is the analysis of generated possible world relationships (filtration [Gol87]). Since a possible world can be characterized by a temporal logic formula, a logic formula transformation can be used as a possible world generator.

In the tableau method implementations, there are common problems.

- During logic formula transformation, negation is expensive.

- Adhoc simplification of logic formula makes a verifier complex.
- Graph operations (required in analysis stages) can be expensive.
- An estimation of required time/space is difficult.

We use deterministic tableau expansion for possible world generation, which makes it possible to avoid repeated determinisation. For possible world characterization the compact binary subterm tree technique is used.

2.3 Model Checking

Model Checking [MC83] is another way of verification. In this case an implementation is given as a possible model for a temporal logic formula. Model checking checks the satisfiability of the formula in the model. In a sense, a theorem prover includes Model Checking. Of course, the Model Checking is limited to the finite length interval, since we only handle finite length intervals in ITL verifier. But we allows arbitrary length. There are three ways to do Model Checking within ITL verifier.

The first method is using ITL formula as a representation of the implementation. Using sufficient numbers of variables, it is easy to write a state transition or finite automaton in ITL. For an ITL formula S and the model representing ITL formula M, the Model Checking is a test of the satisfiability of $S \wedge Q$. But this requires many variables and it makes verification difficult.

The second method is using constraint on the deterministic tableau expansion. In this case, the implementation is represented as a finite automaton. We investigate allowable transition of the finite automaton during the deterministic tableau expansion. In this case, the procedure is easier than checking satisfiability of the specification formula.

The third method is applicable to the BDD implementation. In the fixpoint operation we can simply take conjunction with Characteristic Function (CF) of the implementation. The cost of computation depends on both CFs (the specification and the implementation).

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

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

This $@ - normalform$ represents a non-deterministic state transition shown in Fig.4.

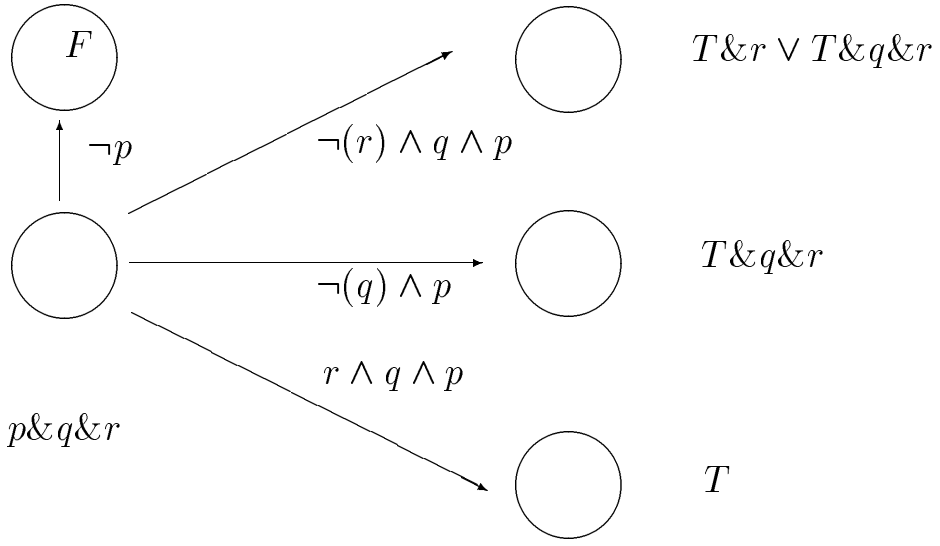


Figure 4: State Transiton for Chop Operator

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,

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

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:

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

$\rightarrow$ 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.

T

$$\begin{aligned} empty(T) &= T \\ more(T) &= @T \end{aligned}$$

$P \wedge Q$

$$\begin{aligned} empty(P \wedge Q) &= empty(P) \wedge empty(Q) \\ more(P \wedge Q) &= more(P) \wedge more(Q) \end{aligned}$$

$P \vee Q$

$$\begin{aligned} empty(P \vee Q) &= empty(P) \vee empty(Q) \\ more(P \vee Q) &= more(P) \vee more(Q) \end{aligned}$$

$\neg P$

$$\begin{aligned} empty(\neg P) &= \neg empty(P) \\ more(\neg P) &= \neg more(P) \end{aligned}$$

$@P$

$$\begin{aligned} empty(@P) &= F \\ more(@P) &= @P \end{aligned}$$

$P\&Q$

$$\begin{aligned} empty(P\&Q) &= empty(P) \wedge empty(Q) \\ more(P\&Q) &= (empty(P) \wedge more(Q)) \vee (more(P)\&Q) \end{aligned}$$

$\exists y Q$ $\quad y$ is removed from leaf conditions

$$\begin{aligned} empty(\exists y Q) &= (empty(y \wedge Q) \vee empty(\neg y \wedge Q)) \\ more(\exists y Q) &= \exists y((more(y \wedge Q) \vee more(\neg y \wedge Q))) \end{aligned}$$

$*(P)$

$$\begin{aligned} empty(*(P)) &= empty(P) \\ more(*(P)) &= more(P)\& *(P) \end{aligned}$$

$P\ proj\ Q$

$$\begin{aligned} empty(P\,projQ) &= empty(Q) \\ more(P\,projQ) &= more(P)\&(P\,projmore(Q)) \end{aligned}$$

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

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

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

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

For the non-empty condition,

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

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 [Wol82], 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.
- $\exists xQ$, 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,

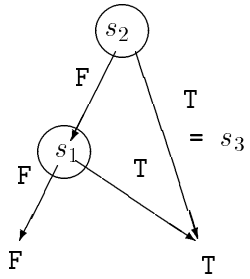
$$s_1 = T\&?(p, F, T).$$

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. 5),

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



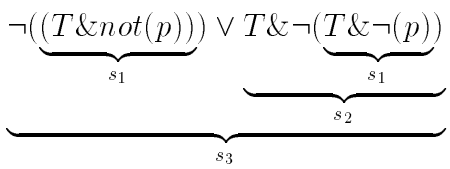


Figure 5: Binary Subterm Tree

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:

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

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

It is translated into @-normal form.

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

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:

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

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

— ?- 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 6: Example of Counter Example Generation

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 $fin(p)$.

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. 7.

```
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 7: A Possible Execution

The implementation includes X window Interface as shown in Fig. 8.

4.2 Why this is better than the Automaton Method?

Since the deterministic tableau method generates deterministic finite automaton from non-deterministic description of ITL formula, it must include determinisation. But unlike the Automaton Method, the determinisation is done by the outer most loop, so that it works faster. It also generates finite automaton even for valid formulae, which represent the possible world structure of the formulae.

However, this requires exponential computation depending on the number of variables and the depth of the temporal logic operators. But this computation is linear to the number of the generated states. Hence it works well for the complex formulae if it represents simple automaton.

The current implementation has been written in Prolog (s.t. C-Prolog or SICStus Prolog) and is very short (730 lines).

Standard forms are used for both the conditional part (classical propositional logic formula) and labelling formula (binary subterm tree).

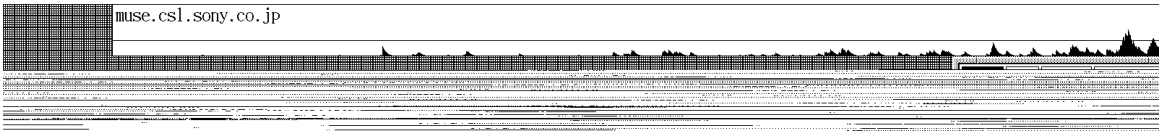


Figure 8: X window display

In fact the Binary Decision Diagram (BDD) [Bry86] is applicable for both the conditional part and the binary subterm tree. In this case, we can replace the determinisation loop with the fixpoint operation.

5 BDD Implementation

The Binary Decision Diagram is a concise representation of boolean formulae. It is a binary decision tree with shared subtrees with fixed ordered variables. Various boolean operations (including fixpoint operation) can be done in an efficient way.

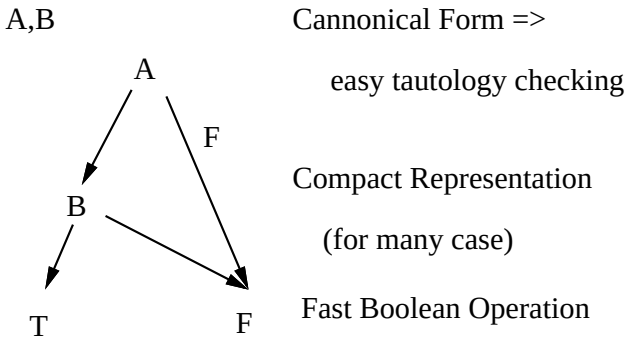


Figure 9: Binary Decision Diagram

Like the deterministic tableau method, BDD implementation has two stages. First, a characteristic function is generated from the ITL formula. This stage is sound being based the ITL axiom. Then we need a fixpoint operation on the characteristic function.

5.1 Generating The Characteristic Function from an ITL formula

Characteristic Function (CF)

$$f(P, E, S, S') \rightarrow \{T, F\}$$

is a boolean formula contains:

P: Local Variable Vector,

E: Empty Flag means termination of an interval,

S: Current Subterm Vector,

S': Next Subterm Vector.

Subterm Vector contains one variable for a numbered label in the subterm tree.

The CF generation is done bottom up. For a local variable x, it contains no numbered subterm nodes. The characteristic function for $P : CF(P)$ contains one element variable vector and zero elements subterm vectors.

$$CF(P) = f([x], E, [], []).$$

To calculate disjunction, a simple boolean operation is enough. The subterm vector and the variable vector are merged incrementally.

$$(P \wedge Q) = f(P, E, S, S') \wedge f(Q, E, S, S')$$

For $P\&Q$, $f_p(P, PE, S_p, S'_p)$, $f_q(Q, E, S_q, S'_q)$,

$$\begin{array}{lcll} f(P\&Q, E, S_1, S'_1) & = & \exists E_P(f_p \wedge f'_q), & \\ where & f'_q & = & (E_P \wedge f_q) \vee (\neg E \wedge \neg E_P \neg R) \\ & S_1 & = & \exists E_P S_p :: S_q :: \exists E_P, f'_q \\ & S'_1 & = & S'_p :: S'_q :: R, R \Leftrightarrow \exists E_P f'_q \end{array}$$

where $P :: Q$ means merged vector of P and Q.

This Characteristic Function represents the relationships among possible worlds which are labelled by boolean formulae on subterm vector and variable vector.

5.2 Fixpoint Operation on Characteristic Function

The original ITL formula has a numbered node C_0 in the binary subterm tree. Therefore, the initial state of fixed point operation is C_0 node. The current possible subterm state is represented by a boolean formula C_n and the next possible state is represented by a boolean formula on S_{n+1}. These are constrained by the Characteristic Function, in this way:

$$C_{n+1} = \exists P \neg E \wedge C_n \wedge f(P, E, S, S').$$

We cannot use the condition $C_n = C_{n+1}$ here because a subterm state can vibrate. The fixpoint is achieved by repeating until $\bigvee_n C_n = \bigvee_{n+1} C_{n+1}$. Since the disjunction operation is monotonic and has a limit, it must terminate eventually.

If $\exists P \neg(E \wedge C_n)$ is satisfiable for some n, this is the counter example.

6 Evaluation of Two Implementations

The BDD version is implemented using the Boolean Constraint Solver in SICStus Prolog. It is also very small (500 lines), but it does not support the existential quantifier, closure and projection.

Unfortunately this BDD version is slower than the Prolog version of deterministic tableau expansion. This is because SICStus Prolog does not support destructive assignment of BDD term, we need a copy operation in the fixpoint operation (which is very slow).

If we use BDD package this method will execute faster than Prolog implementation of the deterministic tableau expansion. But BDD version still requires exponential computation both on the number of the variables (in BDD construction) and the depth of the temporal logic operator (in Fixpoint operation).

7 A Complete ITL Axiom Set

To make a complete ITL Axiom set, we need two steps. First we have to prepare the Axiom which need in order to prove the soundness of tableau expansion (this is easy). We use classical inference rules for propositional logic.

PT. If P is an instance of a propositional tautology then $\vdash P$.

MP. if $\vdash P$ and $\vdash P \Rightarrow Q$ then $\vdash Q$.

Here is a set of Axiom schema. The variables in these formulae are all of second order.

$$\vdash \; empty \& P \Leftrightarrow P$$

$$\vdash \; P \& empty \Leftrightarrow P$$

$$\vdash \; @P \Rightarrow \neg empty$$

$$\vdash \; \bigcirc P \Leftrightarrow empty \vee @P$$

$$\vdash \; @(P \wedge Q) \Leftrightarrow @P \wedge @Q$$

$$\vdash \; @(P \vee Q) \Leftrightarrow @P \vee @Q$$

$$\vdash \; (P \vee Q) \& R \Leftrightarrow (P \& R) \vee (Q \& R)$$

$$\vdash \; (@P) \& Q \Leftrightarrow @(P \& Q)$$

$$\vdash \; empty \wedge (P \& Q) \Leftrightarrow empty \wedge P \wedge Q$$

$$\vdash \; local(P) \Rightarrow (P \wedge Q) \& R \Leftrightarrow P \wedge (Q \& R)$$

$$\vdash \; (P \vee Q)\, proj\ R \Leftrightarrow (P\, proj\ R) \vee (Q\, proj\ R)$$

$$\vdash \; P\, proj\ @Q \Leftrightarrow$$
$$((\neg\ empty \wedge P) \& (P\, projQ)))$$

$$\vdash \; P\, proj\ (Q \vee R) \Leftrightarrow$$
$$(P\, projQ) \vee (P\, projR)$$

$$\vdash \; local(Q) \Rightarrow P\, proj\ (Q \wedge R) \Leftrightarrow$$
$$(P\, projQ) \wedge (P\, projR)$$

$$\vdash \; P\, proj\ empty$$

We may now prove some easy theorems, for example, $\vdash \Diamond empty$. The formula has a chop standard form,

$$T \& empty.$$

Since replacing P to T in

$$\vdash P \& empty \Leftrightarrow P,$$

we have $\vdash T \Rightarrow T \& empty$. Considering the MP inference rule, we have $\vdash T \& empty$. QED.

All these axioms are sound. The proofs are essentially same as the proof of the tableau expansion rule. For example, it includes distribution of disjunction for all temporal logic operators. Using these axioms, a proof sequence for the expansion part of the tableau method is constructed.

The second part is an inference rule for analyzing part of the tableau method.

Loop Eliminator. If we have a set of formula P_i, and for all P_i, there exists P_j,

$$\vdash P_i \Leftrightarrow empty \vee @P_j,$$

then for all P_i, $\vdash P_i$.

In this way we have a simpler axiom system for ITL than choppy logic. This is because we assume finite intervals as stated in choppy logic paper.

8 Future Works

We are now investigating verification with regular variables (second order ITL).

This expansion can be also applicable to first order local ITL. It makes the executable range larger than the older implementation of Tokio [KAFT85].

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.

References

[Bry86] R. E. Bryant: Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers, **C-35**, 8 (August 1986), 667–691.

[dJ91] Gjalt G. de Jong: An Automata Theroretic Approach to Temporal Logic, Computer Aided Verification, Springer-Verlag (1991), 3rd International Workshop, CAV'91.

[Gol87] Robert Goldblatt: Logic of Time and Computation, Vol. 7 of CSLI Lecture Notes, CSLI (1987).

[Hal88] Roger Hale: Temporal logic programming (1988).

[Hir88] 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 (1988).

[KAFT85] S. Kono, T. Aoyagi, M. Fujita and H. Tanaka: Implementation of temporal logic programming language Tokio, Logic Programming '85, Vol. LNCS-221, Springer-Verlag (1985), Lecture Notes in Computer Science.

[KAFT86] S. Kono, T. Aoyagi, M. Fujita and H. Tanaka: Verification of Temporal Logic Programming Language Tokio, Logic Programming Conference '86 (1986), (in Japanese).

[Kon92] Shinji Kono: Automatic verification of interval temporal logic, 8th British Colloquium For Theoretical Computer Science (1992).

[MC83] 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).

[Mil80] R. Milner: A Calculus of Communicating Systems, Vol. 92 of Lecture Note in Computer Science, Springer-Verlag (1980).

[Mil83] G.J. Milne: CIRCAL: A Calculus for Circuit Description, Integration, **1**, 2 (1983), 121–160.

[Mos83] B.C. Moszkowski: Reasoning about digital circuit, Technical Report No.STAN-CS-83-970, Dept. of C.S. Stanford Univ (July 1983).

[RP86] Roni Rosner and Amir Pnueli: A choppy logic (1986).

[RU71] Nicholas Rescher and Alasdair Urquhart: Temporal Logic, Springer-Verlag (1971).

[Wol82] P. Wolper: Synthesis of communicating processes from temporal logic specifications, Technical Report STAN-CS-82-925, Stanford University (1982).