Execution and verification of 2nd order interval temporal logic

Shinji Kono

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

November 16, 1995

Process is the value of second order interval propositional temporal logic (2ITL here after). Not only temporal relation-ship among events, but also processes are directly defined in terms of temporal logic. Since it can express a negation of a process, it covers different range from process algebra. Process includes finite state machine, fairness, a scheduling mechanism, inverse specification, and various temporal logic formula. In this paper we show 2ITL as a subset of interval temporal logic. 2ITL is undecidable, but through investigations on verification procedures, we find decidable subset of ITL. Its automatic verification is also presented.

1 Interval Temporal Logic as Second Order Temporal Logic

Interval temporal logic (Ref. [12] ITL here after) is investigated by various researchers, but because of its undecidability (Ref. [12]), researches are restricted on Local Interval Temporal Logic (Local ITL here after). In Local ITL, variables represents events depending on a clock period like other Temporal Logics. On the other hand, variables of full set ITL represent series of events in intervals of time. We can think the variables represent processes since it includes possibly an infinite stream of events. This is a new view point of ITL, that is a logic on processes.

In case of classical logic, second order propositional logic is trivial, because the value of the second order variable is either T or F. There are no differences between second order logic and normal logic. But in case of temporal logic, value of second order variable is changed from time to time. It looks like first order logic, but the value of the variable is not a return value of a function, which is determined by arbitrary nested function calls. First order logic is undecidable because first order value can be arbitral nested term. But the value of the second order propositional variable is restricted in the meaning of the temporal logic, which can be a finitely represented state machine. Like a first order logic on enumerable term, a second order temporal logic can be decidable.

Before discussing on verification, we have to discuss on expressiveness of various temporal logic.

2 Expressiveness of Various Temporal Logic

After the discovery of a limitation of Linear Time Temporal Logic (LTTL here after. Ref. [13]), complex hierarchy of temporal logic expressiveness is researched. A big surprise is that LTTL is less expressive than ω-Regular expressions on logic formula. We can prove similar limit on Local ITL (Appendix). But we find such a limitation is not serious.

The limit is easily solved by adding quantifier or adding closure operation on logics. Quantified Propositional Temporal Logic has ω-Regular model (Ref. [13]). Actually it is quite easy to construct a representation of arbitrary finite state machine (here after FSM) using quantified variables. We can use quantified variables for 1-hot encoding of a FSM (Ref. [9]). A closure operator $*$ is also sufficient to construct a FSM by using well-known algorithm on Regular Expression (Ref. [7]). A closure operator $*P$ means finite or infinite repetition of a temporal logic formula P. This is easily demonstrated by an example called *evenp*.

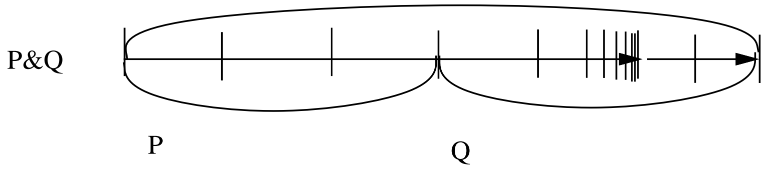
$evenp(p)$ means p is true on every even clock period. We use small letters p, q, r for event variables. In Regular Expression, `(p T)*`. It is proved $evenp(p)$ is not expressed in LTTL nor Local ITL (Appendix). But it is easily expressed by QPTL;

$$evenp(p) \equiv \exists q, q \wedge \Box ((q \rightarrow \bigcirc \neg q) \wedge (\neg q \rightarrow \bigcirc q) \wedge q \rightarrow p).$$

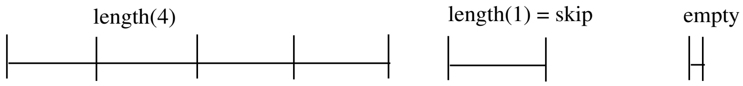
It is also easily expressed by closure operator $*$ like this;

$$evenp(p) \equiv *(((p \wedge length(1)) \& length(1)) \vee \mathrm{empty}).$$

& is a chop operator which is one clock overlapped concatenation of two intervals. Intervals are finite or infinite clock period.



empty means the length of the interval is 0.



But on these extensions, verifications on the logic requires exponential computational complexity on the length of formula. Polynomial order or lower complexity is preferable of course, but recent researches on BDD based verification (Ref. [2]) show we can still verify useful examples even in case of exponential computational complexity, if we takes enough effort to keep state space representation small.

From the view point of automatic verification, closure operator is superior. Because quantifier generates exponential computation on number of variables, which corresponds number of edge of states. Even classical propositional quantified logic requires P-space complexity. In case of closure, it generates exponential computation on depth of temporal logic operator, which increases number of states. It is better than the increase of number of edges.

3 Expressiveness and Decidability of Interval Temporal Logic

ITL and LTTL both feature discrete time, but ITL has the end of the time. Unlike LTTL, ITL has a model on top of Regular expression not on top of ω-Regular expression. In another words, ITL has compact discrete time. It has pros and cons. ITL cannot express fairness like LTTL. For example, $\Box\Diamond p$ does not means p is true infinitely many times but it means p is true on the end of the interval. In fact,

$$\Box\Diamond P \leftrightarrow fin(P)$$

is a theorem in Local ITL. Here we use capital letter P, Q, R (other than F and T) for second order variable or interval variable. Here $fin(P)$ means P is true on the end of the interval and these are defined in Local ITL using & as follows;

$$\begin{array}{lcl} fin(P) & \equiv & (T\&(\text{empty} \wedge P)) \\ \Box\ P & \equiv & \neg(T\&\neg P) \\ \Diamond\ P & \equiv & T\&P \end{array}$$

The theorem is not trivial but it is easy to prove and it means the lack of fairness in Local ITL. But this makes a verification procedure easier, because we don't have to check state loop after tableau expansion as in Müler automaton. If we have no false on the leaves of the FSM, the formula is valid.

Now we know Local ITL with closure has Regular Expression expressiveness. We also have a FSM generation algorithm for Local ITL with closure (Ref. [9]). This shows

Theorem 1 *Local ITL with closure has exact Regular Expression expressiveness.*

In case of full ITL, things goes badly. It is quite easy to show ITL can express Context Free Grammar and Context Dependent Grammar. An interval variable contains a series of states which is a mapping of event variables. We can use an event variable mapping as a terminal symbol and an interval variable as a non-terminal symbol in grammar rule. For example,

$$\boxed{\mathrm{a}}\ ((P\&@Q) \rightarrow (R\&@S))$$

means a context dependent grammar rule: $PQ \rightarrow RS$. Here $\boxed{\mathrm{a}}\ P$ means a formula P is true in all sub-intervals of the interval. It is defined as follows;

$$\boxed{\mathrm{a}}\ (P) \equiv \neg(T\&\neg P\&T).$$

This is rarely used in Local ITL specification, but important in 2ITL. Since satisfiability problem on Context Dependent Grammar is undecidable, full ITL is also undecidable. This is another proof of [12].

Theorem 2 *Full ITL is undecidable.*

But closer look of this problem gives us another insights. S. Kimura shows the undecidability of ITL comes from interval variable itself (Ref. [8], in Japanese). An interval variable can be instantiated with an arbitrary finite or infinite sequence, which includes a sequence generated by a context dependent grammar. This is the source of undecidability. We'll investigate this problem more closer from the view point of automatic verification.

Local variables or events are restricted form of interval variable, which has the same value on empty interval of the beginning of the interval. This restriction makes ITL decidable, it is Local ITL. Locality of P is characterized by next proposition;

$$beg(P) \leftrightarrow P\&T$$

where $beg(P) = ((\text{empty}, P)\&T)$. But we can think another restriction on it.

If we restrict interval variables can be instantiated by Regular Expression generated sequence, we have Regular ITL instead of Local ITL. Unfortunately Regular ITL is still undecidable. Remember we can express Context Free Grammar (here after CFG) in ITL. We can construct an ITL formula which checks a CFG is Regular or not. For a given CFG, construct ITL formula, for example, $P \wedge \boxed{\text{a}} Grammer(P)$. If this is satisfiable in Regular ITL, the CFG is Regular. It is known that whether an context free grammar is Regular or not is undecidable (Ref.[7]). So there is no procedure to check whether Regular model for ITL exists or not.

Theorem 3 *Regular ITL is undecidable.*

In this paper, we'll show some other restrictions which makes ITL decidable. The most simple one is length restriction. These restrictions defined in terms of tableau expansion. In some sense, it can be called operational restriction of interval variables. These restrictions are model restrictions. In case of verification on first order logic, they use incomplete decision procedure. It will not terminate or terminate but gives incomplete results. Our logic is sound and complete in terms of automatic verification procedure. But completeness is defined rather ad-hoc way which is defined in operation in the tableau method.

This is something like a restriction on programming methodology. We can program anything on Turing Machine, but not all the program is good one. If we use a restricted programming method, we can extend reliability of the program. The restrictions of interval variables are thought as such restrictions. Since it is a restriction on second order variables, the expressiveness of the logic is still equivalent to Regular Expressions. This corresponds the fact that a programming methodology does not restrict the power of programming.

4 Specification Examples

Since a second order variable represents a mechanism to terminate an interval, it is possible to think it a fairness.

$$\Diamond_S P \equiv S\&P$$

$\Diamond_S P$ means P is eventually true on fairness S, eventuality-S. Unlike fairness in LTTL, many kinds of different fairness can be defined in 2ITL. In fact the value of second order variable is different on each clock period, so it defines different fairness on each clock period. To define clock period independent fairness, a time constant second order variable is necessary. We discuss it later section. We can think this is an abstraction of watch dog timer or counter. The mechanism of the timer can be defined in terms of ITL;

$$\boxed{\text{a}}\ (S = less(2))$$

This means S assures an interval which is less than 2 clocks. $less(n)$ operator can be defined using n-times nested weak next operator; $less(2) \equiv \bigcirc\bigcirc\text{empty}$. This is a simple watch dog timer mechanism.

For another example, using second order variables, we can directly prove axiom schema. Our verifier is based on model construction, so we can verify some axioms. For example;

$$\boxed{a}\ ((\boxed{a}\ (P) \rightarrow P)) \rightarrow (((\diamondsuit\ (\boxed{a}\ (P))) \rightarrow \boxed{a}\ (P))))).$$

This is called Z discreteness (Ref. [5]). This is valid in LTTL, but not valid in ITL.

Of course, we can use second order variables to define a process by recursion like process algebra.

$$\boxed{a}\ (P \rightarrow ((a \wedge \bigcirc P) \vee (b \wedge \neg a \wedge \text{empty}))$$

But we have to consider the restrictions of second order variables in these process representations in case of verification.

If we try to write a practical specification in temporal logic, it is usually non valid formula even if we quantified all variables. This is because there are hidden assumptions on input variables. These assumptions can be abstracted using second order variables. Using the result of tableau expansion, we can analyze the nature of the assumption. In some sense this is kind of reverse specification.

5 Undecidability from a View Point of Tableau Expansion

In this section, we discuss on a tableau method of 2ITL. In Tableau method on discrete temporal logic, temporal logic terms are decomposed into two parts. One part is depending only on current clock period and the other part only depends on next or later interval.

Here we show an example of decomposition of a chop operator. Assume P, Q is already decomposed into empty parts PE, QE, current clock dependent parts PN_i, QN_i and next interval dependent parts PX_i, QX_i in disjunctive normal form.

$$\begin{aligned} P &= PE \wedge \text{empty} \vee \textstyle\bigvee_{i=0}^{k}(PN_i \wedge @PX_i) \\ Q &= QE \wedge \text{empty} \vee \textstyle\bigvee_{i=0}^{k}(QN_i \wedge @QX_i) \end{aligned}$$

Where @ is next operator with ¬empty and it is called strong next. Then $P\&Q$ is decomposed in this way.

$$\begin{aligned} P\&Q = & (PE \wedge Q) \vee \\ & \textstyle\bigvee_{i=0}^{k}(PN_i \wedge @(PX_i\&Q)) \end{aligned}$$

In practical tableau expansion, BDD standard form and deterministic expansion is necessary, but these are discussed in (Ref. [4, 3, 9]).

In case of second order variable or interval variable, the value of the variable is depend on the both begin-time and end-time of the interval. Since we are working on propositional case, it looks like we can replace the value of the variable by T or F. Yes, the value is T or F, but the value depends on the interval (Fig. 1). So we cannot replace the second order variable with T or F as we did in classical logic.

This situation is demonstrated by an example:

$$\boxed{a}(R = \mathit{length}(2))$$

If we replace R by T or F, this example becomes unsatisfiable. But this is satisfiable if we instantiated R by $\mathit{length}(2)$ or equivalent FSM.

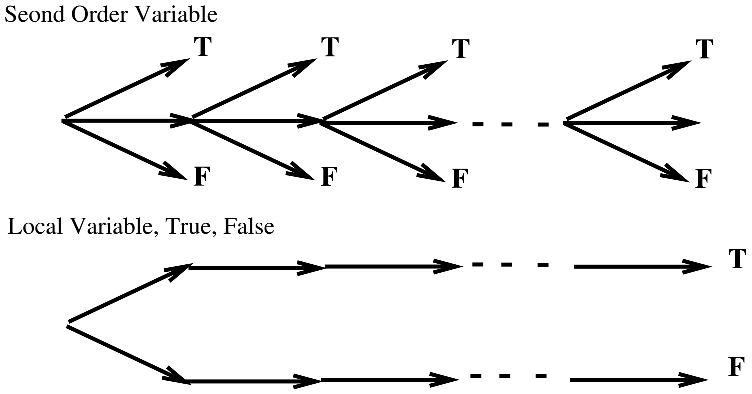
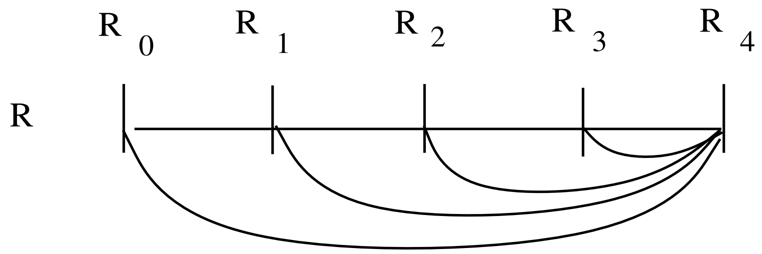


Figure 1: Value of second order variable

If we think R as a FSM, tableau expansion should have next form.

$$\begin{aligned} R &= (\text{empty} \wedge R_0) \vee @R_0 \\ R_n &= (\text{empty} \wedge R_n) \vee @R_{n+1} \end{aligned}$$



R_n is n-th state of R. R_n is also a second order variables. R_n is independent each other if n is different. This is because R_n has different start point and R has different truth value in different interval. Using existential quantifier on second order variable, we can write;

$$R = (\text{empty} \wedge R) \vee @\ \exists S\ S.$$

During the tableau expansions, n increases infinitely. Actually n represents the interval length of R. In case of a formula like $\Box R$, infinitely many R_n are generated. In this way, undecidability of full ITL or 2ITL happens in the tableau expansion.

Projection operator and quantifier on second order variables is not considered in this paper. Since projection operator requires a nest time structure, the time marking of R have to be a nested markings. It looks like possible to implement it, but we have not yet tested yet.

5.1 Length Restriction and Count Restriction

The simplest stopper of the undecidability is length limit.

$$\begin{aligned} R &= \text{empty} \wedge R_0 \wedge @R_0 \\ R_n &= \text{empty} \wedge R_n \wedge @R_{n+1} \text{ if } n < k \\ R_n &= beg(R_n) \text{if } n \geq k \end{aligned}$$

R behaves normal in less than length n interval and after the limit it is fixed to T or F.

R_n is generated on every clock. But if we have only one R_n, we don't have to use specific n. Any other number is also ok. We can compact the number sequence by sort and renaming. But the order of the number have to be preserved. After renaming, n represents number of R_n in a formula. We can call this restriction count limit. The count limit is useful on a formula like this:

$$R\&T.$$

In this example, R_n increases n by 1. This generates a series like this.

$$R\&T, R_1\&T, R_2\&T, ...R_n\&T$$

If we think R and R_1 are equivalent, this example is expanded to itself. Roughly speaking, in count limit method, we have no limit on one time R-eventuality.

5.2 More Complex Restrictions

But above restrictions does not work well on a formula like $T\&R$ that is $\Diamond R$. In this example, expanded formula has a form after n clocks;

$$R_0 \vee R_1 \vee ... \vee R_n \vee T\&R.$$

After R_n reaches the limit, it becomes T or F. In this case, renaming of R_n becomes identity and useless. Looking at the formula carefully, we find every R_n exists only once. The meaning of this formula is not depends on the particular name of R_n and R_n is independent each other. It is possible to remove R_n from the formula. This is called singleton removal.

Singleton removal is possible and it extends possible meanings of second order variables. But currently we have rather complex translation of the formula which generates a lot of states. Besides even after the singleton removal, R still possible to generate increasing complex logical expressions on a formula with multiple occurrence of R_n. 2ITL is of course undecidable with singleton removal. Singleton removal is not practical because it generates big state space. In this reason, we don't investigate singleton removal further here.

Here we show several restriction methods on decidable 2ITL;

length limit Effects of R has time limit,

count limit Number of R is limited,

singleton removal Number of interrelated R is limited.

These are defined in an operational way in the tableau expansion.

To make 2ITL decidable, finiteness of 2ITL term is necessary. What we need is define finite classification of subset of R_n. Here we introduce some of simple classifications, but there can be more useful classifications.

Computational complexity of 2ITL verification is determined by the restriction. Local ITL verification requires exponential complexity of the length of the formula, that mostly comes from determination of expanded states. In the worst case, all combination of the sub terms have to be computed. R_n terms increase the number the sub term, as result in effect of 2^n. In our experience, count limit is slightly faster than length limit.

6 Execution of 2nd Order Interval Temporal Logic

In the tableau expansion based verification (Ref.[9]), a deterministic FSM is generated. This is a method of logic synthesis or program generation. In case of Local ITL, all variables are events. An execution of the FSM is something like flipping traffic rights.

An execution of 2ITL is not so simple. Besides conditions on events variable, it also contains conditions on second order variables. The conditions depend on the restriction methods.

Length limit is the most simple one. It contains two kind of events on R.

termination R is terminated in T or F in less than length n interval.

time out the limit of R expired and results T or F.

These are marks on the FSM and define FSMs for R on each clock period. It also define a trace of R in the execution.

In case of count limit, we have to consider renaming of R_n. Other situation is similar to the length limit case. The renaming is assigned to each transition in the generated FSM. Using renaming information, we can find a FSM definition of R on each clock period.

Since we omit the detail of singleton removal, we cannot discuss execution in singleton removal, here.

7 Time Constant Second Order Variable

In 2ITL, a FSM assigned to a second order variable is varied on the interval. If we assign one fixed FSM on the variable, we have time constant second order variable. The effect of the assignment can be represented using ITL formula, for example,

$$\boxed{\text{a}}\ (R = less(2)).$$

If there is a model for time constant second order variable, there must be a model for non time constant variable. For particular execution path, we can easily check if some of R traces are conflicted. But currently we have no method to find out constant FSM for R.

8 Execution and Verification Examples

In this section, we see actual output of our verifier, which is written in Prolog.

8.1 Example: Length Operator and Second Order Variable

We can demonstrate the difference of length limit and count limit by using simple example: $R \wedge (R = length(10))$. Here we assume limit is 5. $length(10)$ is expressed by nested strong next operator and this becomes the first state.

$$\text{state 1} \quad R \wedge \text{@@@@@@@@@@empty}$$

In case of length limit, it is expanded in this way;

state 2:	$R_1 \wedge$ @@@@@@@@@empty
state 3:	$R_2 \wedge$ @@@@@@@@empty
state 4:	$R_3 \wedge$ @@@@@@@empty
state 5:	$R_4 \wedge$ @@@@@@empty
state 6:	$R_5 \wedge$ @@@@@empty
state 7:	@@@@empty
state 8:	@@@empty
state 9:	@@empty
state 10:	@empty
state 11:	empty

In state 5, R_6's truth value is fixed. R_6 is false entire formula is false otherwise @@@@empty remains. The resulted state diagram is show in (Fig.2). Using this FSM, we can find executions. An execution is shown like this.

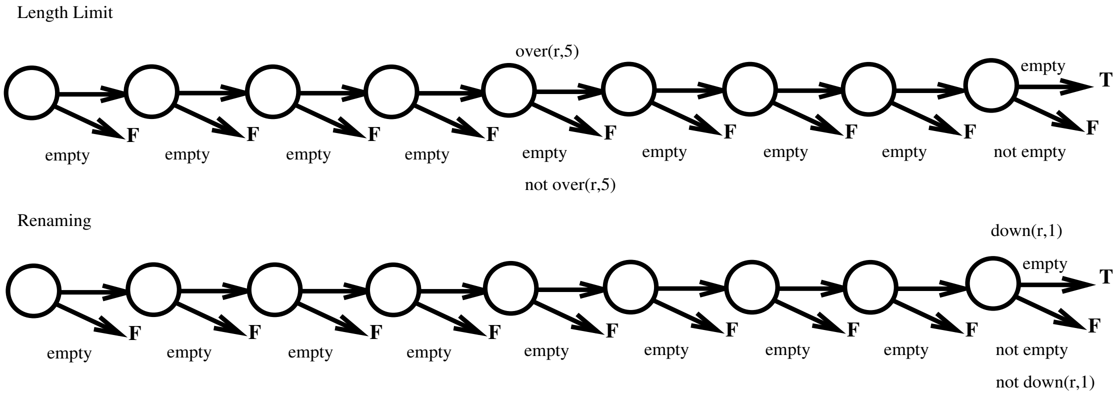


Figure 2: FSM for Length Example

```
| ?- ex((^r,(length(10) = ^r))).
0.46299999999999963 sec.
11 states
16 subterms
23 state transitions
counter example:
0:+r^0    false
| ?- exe.
execution:
0: 2
1: 3
2: 4
3: 5
4: 6
5:+over(r,5) 7
6: 8
7: 9
```

```
8: 10
9: 11
10: 0
yes
```

`^r` is the second order variable notation in our verifier. The first number is clock value and second is the number of state. `+over(r,5)` means R_5 is reached the limit and fixed to T and this is a transition condition of the FSM.

In case of count limit, we have renumbering of R_n. The FSM and execution is shown below. `+r^1` means R_1 is T in the empty interval at the clock. This result looks more correct than previous one.

state 2:	$R_1 \wedge$ @@@@@@@@@empty
state 3:	$R_1 \wedge$ @@@@@@@@empty
state 4:	$R_1 \wedge$ @@@@@@@empty
state 5:	$R_1 \wedge$ @@@@@@empty
state 6:	$R_1 \wedge$ @@@@@empty
state 7:	$R_1 \wedge$ @@@@empty
state 8:	$R_1 \wedge$ @@@empty
state 9:	$R_1 \wedge$ @@empty
state 10:	$R_1 \wedge$ @empty
state 11:	$R_1 \wedge$ empty

```
| ?- ex((^r,(length(10) = ^r))).
0.5179999999999998 sec.
11 states
12 subterms
23 state transitions
yes
| ?- exe.
execution:
0: 2
1: 3
2: 4
3: 5
4: 6
5: 7
6: 8
7: 9
8: 10
9: 11
10:+r^1 0
```

In above examples, limit has no effect on verification. We have 5 clock limit on R, but it does not restrict the length of the interval. The limit restrict the behavior of the variable. The verifier gives us correct FSM if a formula depends only on the limited behavior of second order variable.

$R \wedge \boxed{\text{a}}(length(4) = R)$ is such example. We can write expanded state in a simplified way;

$$\begin{array}{ll}
\text{state 2:} & R \wedge \boxed{\text{a}}(length(4) = R) \wedge \\
& (R_1 \wedge \boxed{\text{i}}(length(3) = R_1) \& T) \\
\text{state 3:} & R \wedge \boxed{\text{a}}(length(4) = R) \wedge \\
& (R_1 \wedge \boxed{\text{i}}(length(3) = R_1) \& T) \wedge \\
& (R_2 \wedge \boxed{\text{i}}(length(2) = R_2) \& T) \\
\text{state 4:} & R \wedge \boxed{\text{a}}(length(4) = R) \wedge \\
& (R_1 \wedge \boxed{\text{i}}(length(3) = R_1) \& T) \wedge \\
& (R_2 \wedge \boxed{\text{i}}(length(2) = R_2) \& T) \wedge \\
& (R_3 \wedge \boxed{\text{i}}(length(1) = R_3) \& T) \\
\text{state 5:} & R \wedge \boxed{\text{a}}(length(4) = R) \wedge \\
& (R_1 \wedge \boxed{\text{i}}(length(3) = R_1) \& T) \wedge \\
& (R_2 \wedge \boxed{\text{i}}(length(2) = R_2) \& T) \wedge \\
& (R_3 \wedge \boxed{\text{i}}(length(1) = R_2) \& T) \wedge \\
& (R_4 \wedge \boxed{\text{i}}(length(0) = R_4) \& T)
\end{array}$$

Here $\boxed{\text{i}}R \equiv \neg(\neg R \& T)$. In this example, $\boxed{\text{a}}$ is decomposed by $\boxed{\text{i}}$.

$$\boxed{\text{a}}P \leftrightarrow \Box\boxed{\text{i}}R.$$

The generated FSM is executed in length 4 interval as we expected.

```
| ?- ex((^r,'[a]'(length(4) = ^r))).
2.362 sec.
7 states
18 subterms
15 state transitions
yes
| ?- exe.
execution:
0:-r^0 2
1:-r^0-r^1 3
2:-r^0-r^1-r^2 4
3:-r^0-r^1-r^2-r^3 5
4:-r^0-r^1-r^2-r^3+r^4 0
yes
```

`-r^n` means R_n is F at the clock period and `+r^n` means R_n is T.

Of course we can prove a theorem like this;

$$\boxed{\text{a}}(R = less(3)) \rightarrow ((R \& R \& R) \rightarrow less(9)).$$

The length limit is working on each R and a complex term like $(R\&R\&R)$ can over come the limit with extra computation. This is because only one R is active at each clock.

```
| ?- ex(('[a]'(^r = less(3)) -> ((^r & ^r & ^r) -> less(9)))).
27.989999999999995 sec.
100 states
35 subterms
629 state transitions
valid
yes
```

The execution of this formula is not interesting because this is a theorem. It succeeds in any interval. To see an interesting one;

```
| ?- ex(('[a]'(^r = less(3)) , ((^r & ^r & ^r)))).
2.519999999999996 sec.
15 states
29 subterms
37 state transitions
yes
| ?- exe(7).
0:+r^0    2
1:+r^0+r^1  3
2:+r^0+r^1+r^2        4
3:+r^0+r^1+r^2-r^3      5
4:+r^0+r^1+r^2-r^3-r^4    6
5:+r^0+r^1+r^2-r^3-r^4-r^5-over(r,5)       8
6:+r^0+r^1+r^2-r^3-r^4-r^5  0
yes
```

Clearly `^r` is T in each interval which length is less than 3 and F otherwise.

As an example of limitation, $R \wedge \boxed{a}(length(10) = R)$ is unsatisfiable if we have length limit 5 or count limit 5. Since this example contains $T\&R$ type expression, there are no big difference between length limit and count limit.

```
| ?- ex((^r,'[a]'(length(10) = ^r))).
3.2110000000000003 sec.
7 states
24 subterms
14 state transitions
yes
| ?- exe.
execution:
unsatisfiable
```

8.2 Example: Z discreteness and Diodorean discreteness

Diodorean discreteness (Ref. [5]) is

$$\boxed{a}(\boxed{a}(((R \rightarrow \boxed{a}R) \rightarrow R))) \rightarrow \diamondsuit\boxed{a}R \rightarrow \boxed{a}R.$$

We can easily prove it under the count limit restriction.

```
?- ex('[a]'('[a]'(((^r->'[a]'(^r))-> ^r)))->'<a>'('[a]'(^r))->'[a]'(^r)).
15 states
24 subterms
98 state transitions
valid
yes
```

In case of Z discreteness;

$$\boxed{a}\ ((\boxed{a}\ (R) \rightarrow R)) \rightarrow (((\Diamond_a\ (\boxed{a}\ (R))) \rightarrow \boxed{a}\ (R)))).$$

We have counter example.

```
?- ex('[a]'(('[a]'(^r)-> ^r))->'<a>'('[a]'(^r))->'[a]'(^r)).
30.710000000000008 sec.
23 states
24 subterms
149 state transitions
counter example:
0:+r^0    2
1:+r^0-r^1  false
```

8.3 Example: Grammar Rules or Recursive Process

A 2ITL formula,

$$R \wedge \boxed{a}((R \rightarrow ((a \wedge @R) \vee (\neg a \wedge b \wedge \mathrm{empty}))))$$

represents a grammar rule or CCS like process. But we are using discrete time and it use $\boxed{a}$ operator, the verifier generates rather big FSM.

```
| ?- ex((^r,'[a]'(( ^r -> ((a,@ ^r); (~a,b,empty)))))).
76.817 sec.
128 states
22 subterms
264 state transitions
yes
| ?- exe(5).
0:+a-r^0 3
1:+a-r^0-r^1 67
2:+a-r^0-r^1-r^2 99
3:+a-r^0-r^1-r^2-r^3 115
4:-a+b+r^0+r^1+r^2+r^3+r^4 0

yes
| ?- exe(10).

no
```

`+a` means `a` is T and `-a` means `a` is F. `exe(10)` requests at least length 10 example and it has no solution because of count limit 5. The solution should be a Regular Expression `a*b`, but we cannot find out the solution because of the undecidability of 2ITL.

In this example, we can replace $\boxed{a}$ by $\Box$. This reduce the number of state to 7 but it gives the same limited execution result.

9 Comparisons

Several methods has been developed for real-time specification.

- Timed automaton (Ref. [1])
- Process Algebra (CCS (Ref.[11]) etc.)
- Higher Order Logic (Ref.[6])
- Temporal Logic (Ref.[5])

Timed automaton can be a basic model of real-time specification. Process algebra provides a program oriented syntax by recursion style. Temporal logic provides natural language like and declarative syntax of specification.

Using closure operator and second order variable, temporal logic effectively includes timed automaton and process algebra. But there are several differences. First, our logic has discrete time model and based on events, that is, every event is synchronized with a global clock. Process algebra has no such global clock. 2ITL itself is not suitable for asynchronous specification but suitable for synchronous or clocked specification. The second different point is there is no summation $+$ in 2ITL. Disjunction can be used as a summation, but it has no mechanism of commitment. Or using Prolog jargon we can say, we have no cut in our language.

In process algebra, all processes are defined by recursions. Propositional temporal logic cannot describe recursions, but 2ITL can. Chop operator's role is very important here. If we use LTTL and Until operator, even if we use second order variable, recursive process is not expressed. But recursion is not a perfect because of our second order variable restriction.

Corresponding part of verification in process algebra is complex hierarchy of bi-simulation. Since 2ITL is logic, failure set and success set is compliment of each other. This corrupts the hierarchy of bi-simulation, and it is defined as temporal logic relation. If we want to check fine grain difference among process including ε events, discrete time can be used. But it requires extra computation on the verification. The other important feature in process algebra, restriction or hiding is expressed by an existential quantifier.

First order logic or higher order logic is also known as powerful tool of specification. Z, ML, HOL and other HDL are successful tools. Of course, formal specification itself is very important, but in these tools not only human aided proof and simulation, automatic verification can be an important part of these systems.

Our verifier can be use as a Model checker (Ref. [10]) also. Actually, a FSM can be used as a part of temporal logic formula. There are two ways to express a FSM in our temporal logic. One is to define the FSM as a new temporal logic operator, the other one is to use a temporal logic formula which represents the FSM. In later case, the translated formula could be big in general, but if the formula is small, it is superior than former method because the verifier can extract information from the structure of the formula. Using these FSM in ITL formula, we can perform a model checking method on the second order interval temporal logic verifier.

10 Future direction

Current method works well on small examples, but we cannot say it is practical. It requires huge computation to verify second order variable and the expressiveness of the variable is restricted. This is because we are concentrated on a FSM based automatic verification and easy restriction.

One possible direction is to construct theorem prover on for example HOL. The other direction is to find out more practical restrictions.

We can extends the restrictions in many way. Actually restrictions on real-time process are easily found in programming technology. An interrupt signal processing or pre-empt processing is usually time limited. Number of preemptable process are limited also. Under these constraints, we can prove the reliability of real-time computation. If we find another useful restrictions, it may become a criteria of verifiable and dependable real-time programming.

The expressiveness of restricted 2ITL is equal to Regular Expression and full 2ITL includes context dependent grammar. One question is this. Is there any restriction which makes 2ITL context free grammar expressiveness?

Extension of this method to projection operator is possible. This is also related to the computational difficulty of singleton removal.

Asynchronous events, edge triggered events or dense time are also future research direction.

Appendix: Local ITL is less expressive than Regular Expression

First we defined Local ITL using mapping function. We use a mapping function $M_{\sigma_0..\sigma_n}(F)$ to define the meaning of Local Interval Temporal Logic (Local ITL) on an interval of time. n can be ω in Local ITL. σ_i means a state of a clock period, which is a mapping of event variable. A state defines truth value mappings of events. $\sigma_0..\sigma_n$ represents a finite or infinite interval.

Local Interval Temporal Logic has

Constants T/F, $M_{\sigma_0..\sigma_n}(T) = T, M_{\sigma_0..\sigma_n}(F) = F$

empty $M_{\sigma_0..\sigma_n}(\text{empty}) = T$ if $n = 0$ otherwise F.

Local variable p, q, r. $M_{\sigma_0..\sigma_n}(p) = M_{\sigma_0}(p) = T/F$. The truth value of a local variable is defined at the first time of the interval.

Chop operator In $\sigma_0..\sigma_n$, there is a state σ_m such that, $M_{\sigma_0..\sigma_n}(F_s \& F_t) = T$ if and only if $M_{\sigma_0..\sigma_m}(F_s) = T$ and $M_{\sigma_m..\sigma_n}(F_t) = T$.

Classical operator Negation and disjunction as usual.

Weak Next $M_{\sigma_0..\sigma_n}(\bigcirc P) = T$ if $M_{\sigma_1..\sigma_n}(P) = T$ or $n = 0$.

Closure operator This is defined in a recursive way. In $\sigma_0..\sigma_n$, there is a state σ_m such that, $M_{\sigma_0..\sigma_n}(*F) = T$ if and only if $M_{\sigma_0..\sigma_m}(F) = T$ and $M_{\sigma_m..\sigma_n}(*F) = T$.

In this proof we don't use closure operator. We also omitted Next operator to make the proof concise. The proof can be extended to include next operator but it may long.

Now we prove $Evenp(p)$ is not expressed in this Local ITL.

We assume a formula P is satisfiable in an infinite interval $\sigma_0..\sigma_\omega$. In case of no next operator case, satisfiability in a non empty finite interval is sufficient.

Using tableau expansion, there is a finite state machine (here after FSM) derived from P. Each state marked by logic formula of sub terms of P. The interval is satisfiable by the FSM. If a FSM can be converted to a form in which at least one state has a next state transition to itself, we call the FSM has single looped state. Using determinization and minization this property can be easily checked.

Now we are going to prove next theorem.

Theorem 4 *If Local ITL formula P is satisfiable in an infinite interval, the derived FSM from P has single looped state in the satisfiable interval state.*

For example, $p\&q$ and $\neg(p\&q)$ generates FSM in Fig 3. The marked state has a transition to itself, and the state is in a satisfiable path. These two FSM has the same form because we use deterministic FSM. In other words, single looped state is preserved in negation. In a special case, F has a single looped state but it has no single looped state on a satisfiable interval. It is also easy to see that single looped state is preserved in chop operator. But a simple formula (empty & empty) does not have single looped states, because it has no out going transitions.

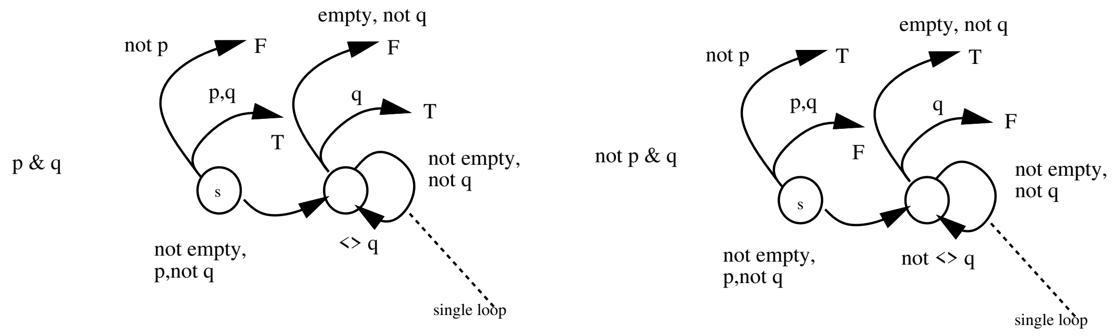


Figure 3: Single Looped State

Proof

First we define polarity. Polarity has a value T or F. Here we use an inductive definition on sub terms of P in top-down. We use P_s, P_t for sub terms of P.

Initial Case P's polarity is T.

Negation Case P has a form $\neg P_s$. The polarity of sub terms P_s is negation of P.

Otherwise Polarity of sub terms has the same polarity of parent term.

Next, we want to prove this. If a formula does not have single looped states, there are positive polarized empty in all positive polarized disjunction leaves, negative polarized conjunction leaves and chop leaves. It is proved by another bottom up induction on ITL formula.

Initial Case If P is local variable, T or F, it has a single looped state.

Initial Empty Case Positive polarized empty does not have single looped states. Negative polarized empty has a single looped state.

Chop Case P has a form $P_s\&P_t$. If either P_s or P_t has the a single looped state, P has a single looped state.

Negation Case P has a form $\neg P_s$. If P_s has a single looped state, P has a single looped state. Recall F has a single looped state and the shape of deterministic FSM is preserved in negation.

Disjunction Case P has a form $P_s \vee P_t$. If polarity of P is positive and if either P_s or P_t has a single looped state, P has a single looped state. If polarity of P is negative and if both P_s and P_t has a single looped state, P has a single looped state.

Conjunction Case P has a form $P_s \wedge P_t$. If polarity of P is positive and if both P_s and P_t has a single looped state, P has a single looped state. If polarity of P is negative and if either P_s or P_t has a single looped state, P has a single looped state.

These are easily verified by the semantics of operators and characteristics of deterministic FSM. Roughly saying, the basic case shows we have no single looped states if and only if we have positive emptyin it, and other induction cases show it is preserved in all Local ITL operators (except closure or next).

Assume P does not have a single looped state in each state of the infinite satisfiable interval. Then in all positive polarized disjunction leaves, negative polarized conjunction leaves and chop leaves, all the sub terms have positive polarized empty. This means P is only satisfiable in an empty interval. It contradicts the assumption that P is satisfiable in an infinite interval. **End Proof**

Evenp(p) means p is true at every even clock period. The FSM for *Even*(p) is easy as in Fig. 4 and it does not have single looped state in the satisfiable interval. (But it has a single loop state on the unsatisfiable interval because it contains F leaf) So we cannot express *Evenp*(p) in Local ITL, which is easily written in Regular Expression or QPTL.

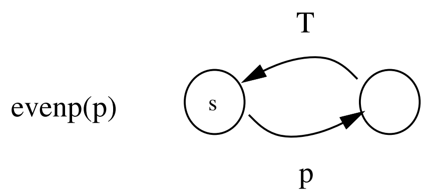


Figure 4: FSM of Evenp

The extension of the proof for including next operator is possible but not so easy. It requires careful handling of empty and next interaction. We can express *Evenp*(p) for a fixed finite interval using next operator, but in case of infinite intervals, there is a sub interval which is generated by non next restricted way. The sub interval has single looped state.

It is easy to show that a closure operator does not keep the single looped state. If we have a quantifier, the form of FSM changes drastically by quantifier and we cannot extend the proof. Since *Evenp*(p) is easily written with a closure or a quantifier, *Even*(p) can be used as a counter example.

Appendix: List of Definitions

$$
\begin{array}{lcl}
fin(P) & \equiv & (T\&(\mathrm{empty} \wedge P)) \\
beg(P) & \equiv & ((\mathrm{empty} \wedge P)\&T) \\
@P & \equiv & \neg\mathrm{empty} \wedge \bigcirc P \\
\Diamond\ P & \equiv & T\&P \\
\Diamond_S P & \equiv & S\&P \\
\Box\ P & \equiv & \neg(T\&\neg P) \\
\Diamond\mkern-13mu{\scriptstyle a}\ \ P & \equiv & T\&P\&T \\
\boxed{\scriptstyle a}\ P & \equiv & \neg(T\&\neg P\&T) \\
less(2) & \equiv & \bigcirc\bigcirc\mathrm{empty} \\
length(2) & \equiv & @@\mathrm{empty} \\
\Diamond\mkern-12mu{\scriptstyle i}\ \ P & \equiv & P\&T \\
\boxed{\scriptstyle i}P & \equiv & \neg(\neg P\&T)
\end{array}
$$

References

[1] Rajeev Alur. The Theory of Timed Automata. In *Real-time: Theory in Practice, LNCS 600*. Springer-Verlag, 1991.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 667–691, August 1986.

[3] Masahiro Fujita and Shinji Kono. Synthesis of Contrllers from Interval Temporal Loigc Specification. In *International Conference on Computer Design*, October 1993.

[4] Masahiro Fujita and Shinji Kono. Synthesis of Contrllers from Interval Temporal Loigc Specification. *International Workshop on Logic Synthesis*, May 23-26, 1993.

[5] Robert Goldblatt. *Logic of Time and Computation*, volume 7 of *CSLI Lecture Notes*. CSLI, 1987.

[6] M.J.C. Gordon and T.F. Melham. Introduction to the hol system. Technical report, Cambridge University, 1994.

[7] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, language and computation*. Addison-Wesley, 1979.

[8] Shinji Kimura and Shuzou Yajima. Formal languages satifing temporal logic formula. *IECE Japan*, Vol. J70-D, No. 1, pp. 117–123, 1987. in Japanese.

[9] Shinji Kono. A Combination of Clausal and Non Clausal Temporal Logic Program. In *Executable Modal and Temporal Logics*, volume LNAI-897. Springer-Verlag, 1994. Lecture Notes in Artifificial Intelligence.

[10] K.L. McMillan. "symbolic model checking: An approach to the state explosion problem". Technical Report CMU-CS-92-131, Carnegie Mellon University, May 1992.

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

[12] B.C. Moszkowski. Executing temporal logic programs. Technical Report 55, Computer Laboratory, Univ. of Cambridge, 1984.

[13] P. Wolper. Temporal logic can be more expressive. In *22nd Annual Symposium on Foundation of Computer Science*, October 1981.