

Synthesis of controllers from Interval Temporal Logic specification

Masahiro Fujita* and Shinji Kono†

February 2, 1993

We present a method which accepts Interval Temporal Logic (ITL) formulas as specification and automatically generates state machines. Since ITL is based on intervals and their sequences, we can easily describe both concurrent and serial behaviors by specifying behaviors inside intervals and sequences among them. ITL formulas are expanded into the ones for the cases that the present state is the final state of the current interval and those for the cases that there is next state in the current interval, which can be easily converted into state machines, by tabulated rules for temporal operators. The specification in ITL can also be used as a constraint for a (possibly non-deterministic) state machine which is an abstraction for an existing sequential circuit. In that case, the final synthesized circuit satisfies the properties both in ITL and the original state machine, which can be useful for redesign or engineering change. The generated state machines can be further processed by logic synthesizer, such as SIS, and transformed into synchronous/asynchronous sequential circuits. We present experimental results and show the usefulness of our method.

1 Introduction

Temporal Logic is an extension of traditional logic with temporal operators, which specify allowed values of variables in multiple time frames. We can specify behaviors of digital systems concisely using temporal logic just like we can specify combinational circuits using traditional logic.

*FUJITSU LABORATORIES LTD. Processor Lab. 1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, JAPAN, fujita@flab.fujitsu.co.jp

†Sony Computer Science Laboratory Inc. Higashi-gotanda, Shinagawa-ku, Tokyo 141, JAPAN, kono@csl.sony.co.jp

There have been many research activities on verification based on temporal logic (examples are [1, 6, 3, 8]) and some are reported on synthesis. Synthesis from temporal logic formulas means the automatic generation of state machines from temporal logic formulas. In almost all cases it is based on tableau expansion: the property of temporal logic that any temporal logic formulas can be expanded into ones for the current state and those for the next state [14, 2]. Using this property, we can translate any temporal logic formulas into state transition forms.

However, all methods reported so far use Linear Time Temporal Logic (LTTL in short and its extension, Extended Temporal Logic, ETL in short) or Branching Time Temporal Logic (BTTL in short). These logics have temporal operators such as, *always*, *eventually*, *next*, and *until*. These operators are useful when we describe concurrent behaviors, such as handshaking protocols, it is not easy to describe serial behaviors, which are common in (serial) programming languages. The latter is necessary when we want to design practical state machines, because almost always digital systems have serial behaviors in some way.

Interval Temporal Logic (ITL in short) is proposed and used to describe digital circuits in [11]. ITL is based on the idea of intervals which are collections of states. Temporal operators in ITL can specify allowed sequences of intervals and also allowed values of variables among the states within an interval. So, we can easily specify both serial and concurrent properties in terms of intervals in ITL.

There are several research activities on ITL. Interpreter and simulator for subsets of ITL are reported in [7, 9]. However, there have been no literature on the synthesis from ITL formulas. This is because there have been no tableau methods which can handle ITL formulas completely and efficiently.

In this paper, we present a tableau expansion method [8], assuming that every interval is finite, i.e., every interval must terminate. Using this assumption, we can directly expand any ITL formulas into the ones for the case that the present state is the final state of the current interval and those for the case that there is next state in the current interval. The resulting ITL formulas form state transition relations, which can be further processed by logic synthesizer, such as SIS.

The presented method can also be used to add a constraint to an existing design. In other words, we can specify the constraint in ITL which we want to add to an existing design in state machines and then synthesize circuits using that existing design and ITL formulas, which is a kind of rectification or redesign of sequential circuits. Similar situations can happen in high-level synthesis where the detailed scheduling is not fixed and in the synthesis of sequential circuits with don't care sequences [12]. In both cases, designs have non-determinisms which can be restricted by adding ITL formulas for the constraints to the designs.

We have implemented the above method using Prolog on PC notebook. This implementation demonstrates that ITL formulas much larger than trivial ones can be processed

within a practical time, even if PC notebook is used.

This paper is organized as follows. In section 2, we introduce ITL and show how to specify serial and concurrent behaviors in ITL. In section 3, we present the expansion method which generates state machines from ITL formulas. In section 4, we present a redesign method for sequential circuits using ITL. Section 5 gives some experimental results and section 6 is a concluding remark.

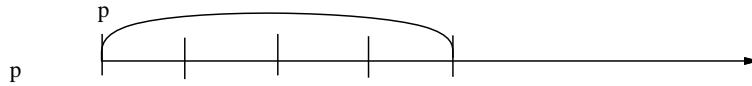
2 Interval Temporal Logic

Interval Temporal Logic[11] (ITL in short) 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). First we show informal visual representation of basic operators in ITL. Here we assume that there is a minimum unit of time and clock is working on that time unit. System can change its internal state at each clock. Thus, we use the terms clock and state interchangeably.

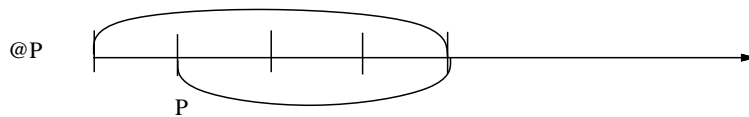
An interval can be viewed as a finite line which has number of clock ticks. An operator *empty* is true on the length 0 interval, and *length* specifies the length of that interval.



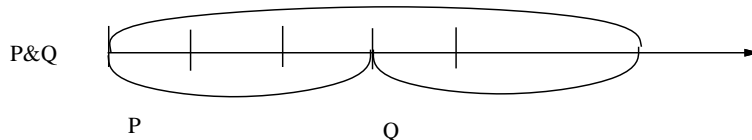
A local (or atomic) variable p means p occurs at the beginning of the interval.



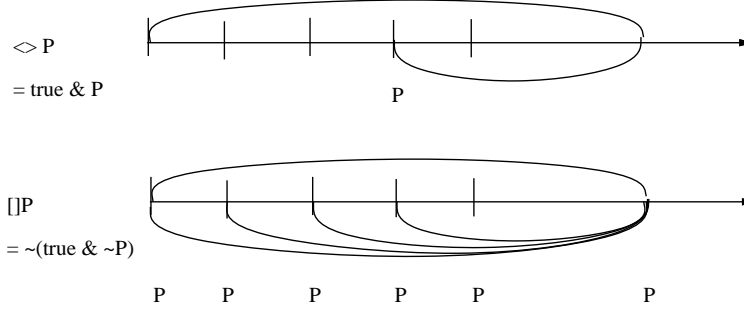
The *next* operator $@P$ means P becomes true after one clock cycle (or in the next state). Thus, in ITL, $@P$'s interval must be one clock cycle longer than P 's and $@P$ is false on the empty interval. We call this *strongnext*. We write *weaknext* $\circ P$ as $@P \vee empty$ where P can be any temporal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



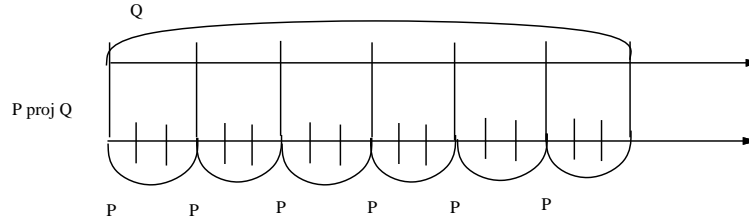
We introduce the chop operator ‘&’ which combines two intervals. $P&Q$ roughly means “do P then Q ”.



Using the chop operator we can express sometime \diamond and always \square .



A projection operator creates coarse grain time using a repeated interval. $P \text{ proj } Q$ 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 formulas and small letter p, q, r for propositional variables.

$$\begin{aligned}
 M_{ij}(T) &= T(\text{true}) \\
 M_{ij}(F) &= F(\text{false}) \\
 M_{ij}(p) &= \forall j. \text{ a variable } p \text{ has a value } T \text{ or } F \\
 M_{ij}(P \Rightarrow Q) &= T \text{ when } M_{ij}(P) = F \text{ or } M_{ij}(Q) = T \\
 &F \text{ otherwise} \\
 M_{ij}(\exists p. f(p)) &= \text{for a free variable } p \text{ in } f(p), \text{ there is a mapping} \\
 &\text{function } M' \text{ which contains a variable } p \\
 &M'_{ij}(f(p)) = T \\
 M_{ij}(P \& Q) &= T \text{ when } i \leq \exists k \leq j, M_{ik}(P) = T, M_{kj}(Q) = T \\
 &F \text{ otherwise} \\
 M_{ij}(@P) &= T \text{ when } i + 1 \leq j, M_{(i+1)j}(P) = T \\
 &F \text{ otherwise} \\
 M_{ij}(\text{empty}) &= T \text{ when } i = j
 \end{aligned}$$

F otherwise

$M_{ij}(P \text{ proj } Q) =$ there is a length n ordered sequence k of integer such as,

$$0 \leq \exists n \leq j - i, \forall r \text{ in } 0..n - 1, i \leq k_r \leq j, k_r < k_{r+1}$$

This sequence is constrained by P,

$$M_{k_r k_{r+1}}(P) = T$$

Q is true on the coarse grain time,

$$\forall s \text{ in } 0..n \text{ there are some mapping functions } M'_{rs} = M_{k_r k_s}$$

$$M'_{0n}(Q) = T$$

$$M_{ij}(\text{local}(P)) = T, \text{ if } i \leq \forall k \leq j, M_{ik}(P) = M_{ii}(P)$$

$\text{local}(P)$ means P is independent of the termination clock period (i.e. P is only 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,

$$P \vee Q \equiv (\neg P) \Rightarrow Q$$

$$P \wedge Q \equiv \neg(P \Rightarrow \neg Q)$$

$$P \Leftrightarrow Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

$$\text{more} \equiv \neg \text{empty}$$

$$\diamond P \equiv T \& P$$

$$\square P \equiv \neg \diamond \neg P$$

$$\bigcirc P \equiv @P \vee \text{empty}$$

$$\text{skip} \equiv @\text{empty}$$

$$\text{length}(n) \equiv \underbrace{@@ \dots @}_n \text{empty}$$

$$\text{less}(n) \equiv \underbrace{\bigcirc \bigcirc \dots \bigcirc}_n F$$

$$\forall P f(P) \equiv \neg \exists P \neg f(P)$$

$$P \&\& Q \equiv (P \wedge \neg \text{empty}) \& Q$$

$$* P \equiv (P \text{ proj } T) \vee (\text{empty} \wedge P) \text{ (closure)}$$

$$\text{fin}(P) \equiv \text{empty} \Rightarrow P$$

$$\text{halt}(P) \equiv \text{empty} \Leftrightarrow P$$

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.

Here we assume that every interval is finite. This makes a simple theorem, $\diamond \text{empty}$, (every interval must include an termination point). Hence, its dual $\square \text{more}$ is unsatisfiable

since we cannot extend the interval indefinitely. Later we prove that

$$(\Box \Diamond P) \Leftrightarrow (\Diamond \Box P) \Leftrightarrow \text{fin}(P),$$

from which we deduce ITL cannot express fairness. However, the decision procedure is simple for finite intervals.

2.1 Specification in Interval Temporal Logic

In ITL, it is easy to express sequential execution and time out. We can describe a little complex property like:

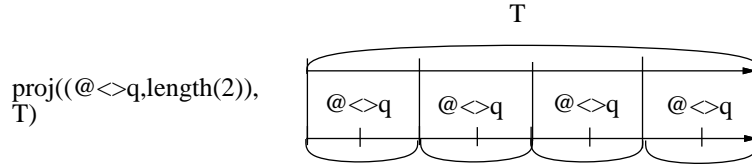
$$((\text{less}(5) \wedge \Diamond p \wedge \Diamond q) \vee (\text{length}(6) \& s)) \& \Box r$$

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. If p and q do not happen within 5 clock cycles, s is happen before r .

Using proj , the repeated event and time sharing task are easily described as in [7]. The expression

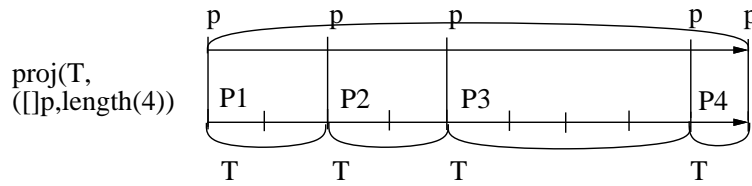
$$(\text{length}(2) \wedge @ \Diamond p) \text{proj } T$$

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



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

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



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

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

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.

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

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

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

This @-normalform represents a non-deterministic state transition shown in Fig.1.

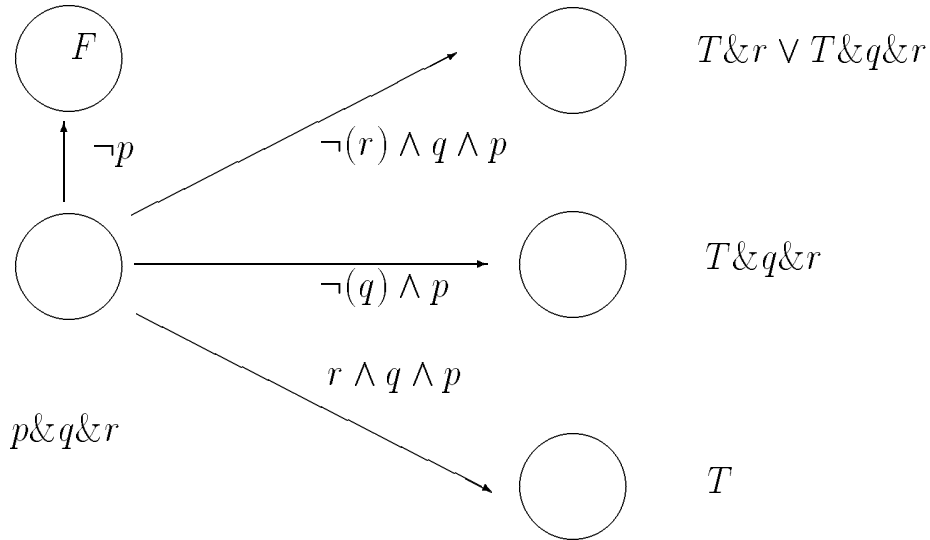


Figure 1: State Transition for Chop Operator

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,

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

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 formulas then non-overlapped conditions can be represented as a binary decision tree, in which leaves are ITL formulas. 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:

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

\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 formulas, 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

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

$P \wedge Q$

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

$P \vee Q$

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

$\neg P$

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

$@P$

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

$P \& Q$

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

$\exists y Q$

y is removed from leaf conditions

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

$*(P)$

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

$P \text{ proj } Q$

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

These transformation rules are part of the complete axiom system in ITL. Soundness of these transformations can be seen by checking the definition of the temporal logic operators.

3.1 Expansion Example

The tableau expansion of $p \& q$ (where p and q are atomic variables) generates a tree with 6 leaves. For the empty condition we can replace $\&$ with \wedge . Then we have

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

For the non-empty condition,

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

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

3.2 Binary Subterm Diagram

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

To overcome this situation, we introduce a binary subterm diagram. This 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. $\&, \text{proj}$)

Translation from ITL formula to binary subterm diagram is done in a bottom-up way. For example, $\diamond \square p$ 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. 2),

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

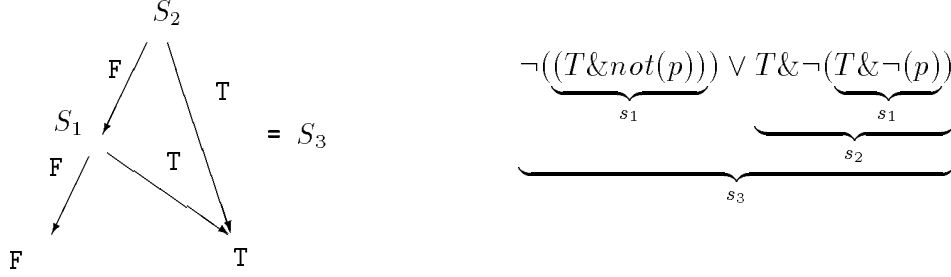


Figure 2: Binary Subterm Tree

In this way, we can store generated formulas compactly in a binary subterm diagram. 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.

In other words, we regard each subterm as an independent variable and generate Binary Decision Diagram using that variable. This results in a canonical form for sub-formulas generated by tableau rules from a given temporal logic formula. So, we can easily check whether the newly generated sub-formulas are those that have been already processed or not. This is one of the key point in our implementation of our synthesis program.

3.3 Termination of tableau expansion

If binary subterm diagrams 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 diagram in the argument of the operator then it may require a new node. However, we can ensure that expansion of a given ITL formula only generate finite variants.

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*,

$$more(P \text{ proj } Q) = more(P) \& (P \text{ proj } more(Q)),$$

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

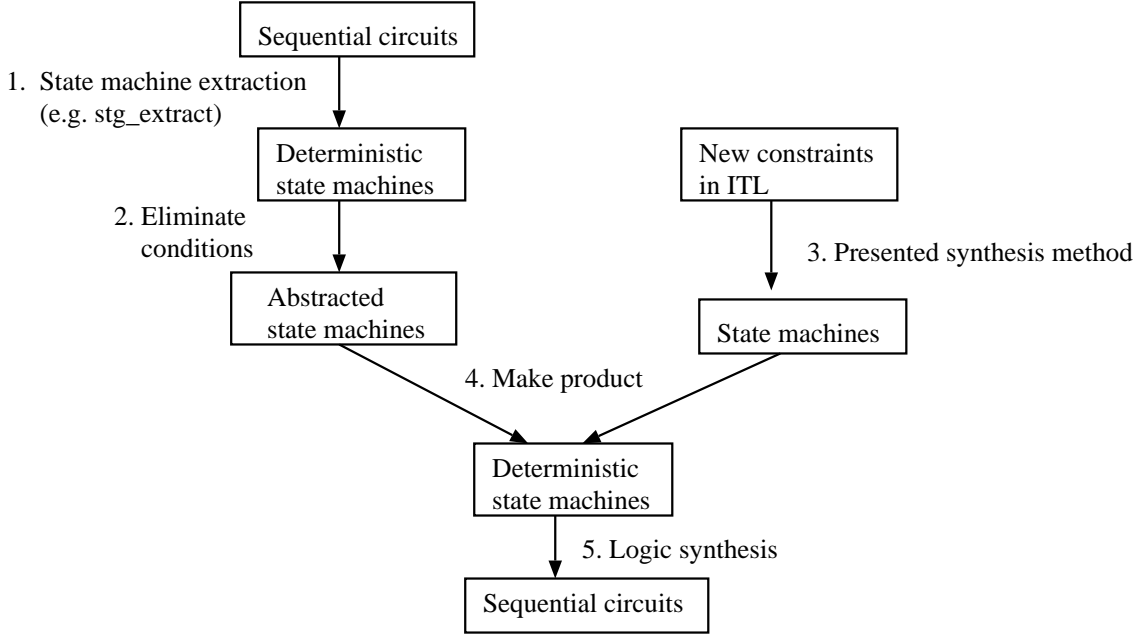


Figure 3: Flow of Our Redesign Method

can vary according to the variant of $more(Q)$. So the number of generated formulas is less than the number of products of variants for $more(P)$ and $more(Q)$.

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

4 A Redesign Method for Sequential Circuits

We can simply use the method which generates state machine representation from any ITL formulas as a procedure to synthesize state machines and hence sequential circuits from ITL formulas. However, we cannot handle very complex ITL formulas within a reasonable time, since the expansion time grows exponentially with the number of temporal operators in the worst case. So, more practical situation is to use ITL formulas as a partial specification for the system being designed. In other words, we assume a (possibly non-deterministic) state machine as a skeleton of the required design and add necessary constraints to it in terms of ITL formals. In this scenario, we do not have to handle very complex ITL formulas. Instead those are represented as state machines abstracted from existing circuits and ITL formulas are only given for the remaining properties which must be satisfied by those state machines.

This redesign method for sequential circuits are shown in Figure 3. The (possibly non-deterministic) state machines can be obtained from existing sequential circuits. First we

extract state machines from given sequential circuits by the procedure such as *stg_extract* in SIS (1. in Figure 3). Then the conditions for variables which must be modified are eliminated from the generated state machines. This process generates possibly non-deterministic state machines in the sense that the eliminated constraints for variables are not sensed and controlled (2. in Figure 3). Then we provide ITL formulas for the specification of the eliminated variables so that they satisfy the required properties. These ITL formulas are expanded into state machines (3. in Figure 3). In this stage, we have two different state machines, one is obtained from the existing sequential circuits, and the other is obtained from ITL formulas. By making the product of these two state machines, the resulting state machine is the one we wanted, i.e., the one which satisfies both the properties in the extracted state machine and the ones in ITL formulas (4. in Figure 3). The final state machine can be logic synthesized and the desired sequential circuit is obtained (5. in Figure 3).

The above is a straight forward way to redesign an existing circuit and the final circuit can have completely different circuit structure, which cannot be affordable in some case. If we want a sequential circuit which satisfy the new constraints but whose structure are similar to the existing circuit, we can use the method presented in [4, 13, 5, 10]. By using these method, we can obtain a final circuit which has a very similar structure, e.g., a circuit having most part of the original circuit, or a circuit having only small extra circuit.

5 Implementation and some results

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 diagram). So ad-hoc simplification is not necessary. This implementation includes X window interface too.

The synthesized state machine for $(length(2) \wedge @ \diamond p) \text{ proj } T$ is shown in Figure 4. The CPU time for this expansion only takes 1.3 seconds on PC notebook with 16MHz 386 chip. As you can see from the figure, some transition has no conditions. This can be considered that the generated machines only satisfy partial requirements. By making the product between this and a state machine which is an abstraction of an existing design, we can get the final design. For example, suppose we have an existing design as shown in Figure 5. In this circuit the output is q . Now suppose we want to modify the output value of q as follows: when transitioning from state S0 to S1 and from S2 to S3, the output value q must be 0. Otherwise, the output q must be one every 2 clock cycles. In this case, first we modify the existing design in Figure 5 to the one shown in Figure 6. Here only the necessary 0 value for q is specified. Then we compute the product of the

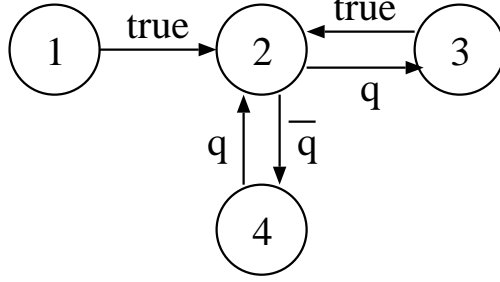


Figure 4: Synthesized state machine for $(length(2) \wedge @ \diamond p) proj T$

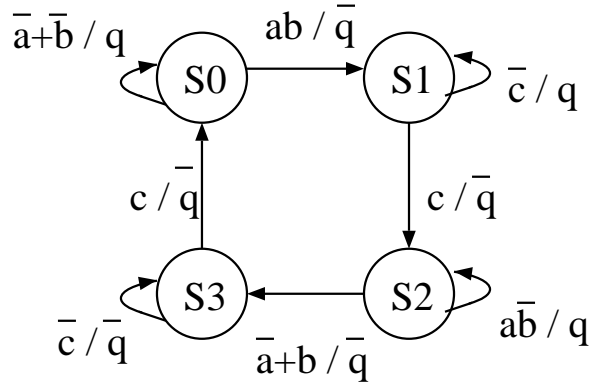


Figure 5: An existing design

machines shown in Figure 4 and Figure 6 and get the final state machine which satisfies the requirement for q . From the state machine we can synthesize the final circuit using the rectification methods shown in [4, 13, 5, 10].

Expansion of a more complex formula,

$$((less(5) \wedge \diamond p \wedge \diamond q) \vee (length(6) \& s)) \& \square r$$

takes 17.7 seconds on PC notebook to be expanded. The number of the generated states is 28. As can be seen from this expansion, rather complex formulas can be expanded within practical time on PC notebook.

We will show practical size examples including the synthesized sequential circuits in the final version of the paper.

6 Conclusions

We have presented a synthesis method which generate sequential circuits from ITL formulas. We have also shown a redesign method for sequential circuit which has, we believe,

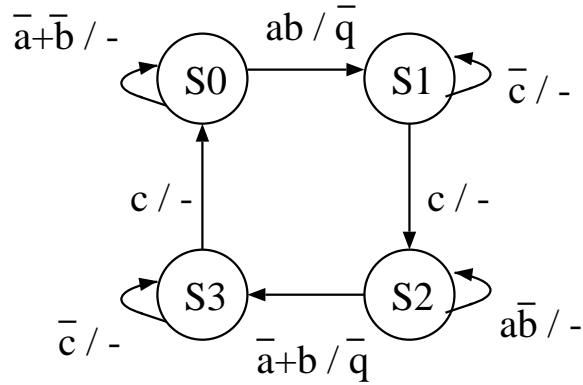


Figure 6: Modified design: some conditions for output q are eliminated

practical values in real designs.

Although our method requires exponential computation in terms of the number of variables and the depth of the temporal logic operators, its computation is linear to the number of the generated states due to binary subterm diagrams. Hence it works well for the complex formulas if it does not include many variables and temporal operators, which are common cases when we apply our method to redesign.

Our program can work and process practical examples on PC notebook. So, if we install logic synthesizer, such as SIS, on PC notebook, we can easily do redesign for sequential circuits out of office. This can be important, since redesign can be a collection of many trial and errors, and designers may want to do them at home.

References

- [1] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. "Symbolic model checking: 10^{20} states and beyond". In *Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science*, Jun. 1990.
- [2] E.M. Clarke and E.A. Emerson. "Synthesis of Synchronization Skeletons from Branching Time Temporal Logic". In *Proc. of the Workshop on Logics of Programs, LNCS-131, Springer-Verlag*, 1982.
- [3] G.G. de Jong. "An Automata Theoretic Approach to Temporal Logic". In *Proc. of Computer Aided Verification*, Jul. 1991.
- [4] M. Fujita, T. Kakuda, and Y. Matsunaga. "Redesign and automatic error correction of combinational circuits". In *Proc. of IFIP Working Conference on Logic and Architectural Synthesis*, May 1990.

- [5] M. Fujita, Y. Tamiya, Y. Kukimoto, and K.C. Chen. Application of Boolean Unification to Combinational Logic Synthesis. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pp. 510–513, Nov. 1991.
- [6] M. Fujita, H. Tanaka, and T. Moto-oka. “Logic Design Assistance with Temporal Logic”. In *Proc. of IFIP WG10.2 International Conference on Hardware Description Languages and their Applications*, Aug. 1983.
- [7] R. Hale. “Temporal Logic Programming”. Technical Report PhD thesis, Computer Laboratory, Cambridge University, 1988.
- [8] S. Kono. “Automatic verification of interval temporal logic”. In *Proc. 8th British Colloquium for theoretical computer science*, Mar. 1992.
- [9] S. Kono, T. Aoyagi, M. Fujita, and H. Tanaka. “Implementation of temporal logic programming language Tokio”. In *Proc. Logic Programming Conference, LNCS-221, Springer-Verlag*, 1985.
- [10] Y. Kukimoto and M. Fujita. “Rectification Method for Lookup-Table Type FPGA’s”. In *Proc. of ICCAD-92*, pp. 54–61, Nov. 1992.
- [11] B. Moszkowski. “Reasoning about digital circuits”. Technical Report STAN-CS-83-970, Dept. of Computer Science, Stanford University, Jul. 1983.
- [12] J-K. Rho, G. Hachtel, and F. Somenzi. “Don’t care sequences and the optimization of interacting finite state machines”. In *Proc. of EDAC-91*, pp. 418–421, Feb. 1991.
- [13] Y. Watanabe. “Minimization of Multiple-Valued Relations”. Technical Report UCB/ERL M91/48, Electronics Research Laboratory, University of California, Berkeley, May 1991.
- [14] P. Wolper. “Synthesis of communicating processes from temporal logic specifications”. Technical Report STAN-CS-82-925, Dept. of Computer Science, Stanford University, 1982.