Synthesis of controllers from Interval Temporal Logic specification

Masahiro Fujita[*] and Shinji Kono[†]

We present a method which accepts Interval Temporal Logic (ITL) formulas as specification and automatically generates state machines. The specification in ITL can also be used as a constraint for a state machine which is an abstraction for an existing sequential circuit, which can be useful for redesign or engineering change. The generated state machines can be further processed by logic synthesizer, such as SIS. We present experimental results and show the usefulness of our method. `Keyword:` temporal logic, logic synthesis

1 Introduction

Temporal Logic is an extension of traditional logic with temporal operators, which specify allowed values of variables in multiple time frames. There have been many research activities on verification based on temporal logic, [1, 5]). Interval Temporal Logic (ITL in short) is proposed and used to describe digital circuits. 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.

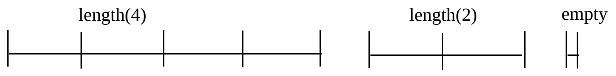
In this paper, we present a tableau expansion method [3, 2], with BDD technology. Our method is more efficient than automaton method [1] and practical subset of [5]. The presented method can also be used to add a constraint to an existing design. Similar situations can happen in high-level synthesis where the detailed scheduling is not fixed and in the synthesis of sequential circuits with don't care sequences. In both cases, designs have non-determinisms which can be restricted by adding ITL formulas for the constaints 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.

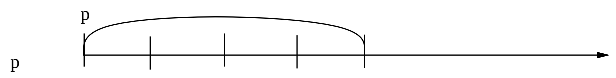
This paper is organized as follows. In section 2, we introduce ITL and show how to specify serial and concurrent behaviors in ITL. In section 3, we present the expansion method which generates state machines from ITL formulas. In section 4, we present a redesign method for sequential circuits using ITL.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

2 Interval Temporal Logic

ITL uses a sequencing modal operator as its basis. In this logic, it is very easy to express control structures in conventional programming languages. Here we show informal visual representation of basic operators in ITL. 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.



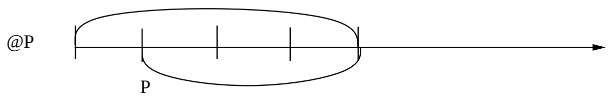
A local (or atomic) variable p means p occurs at the beginning of the interval.



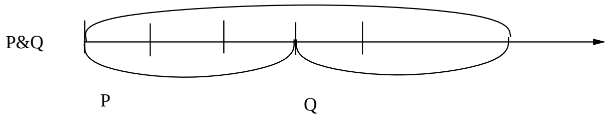
The $next$ operator $@P$ means P becomes true after one clock cycle (or in the next state). Thus, in ITL, $@P$'s interval must be one clock cycle longer than P's and $@P$ is false on the empty interval.

[*]FUJITSU LABORATORIES LTD. Processor Lab. 1015 Kamikodanaka, Nakahara_ku, Kawasaki 211, JAPAN, fujita@flab.fujitsu.co.jp

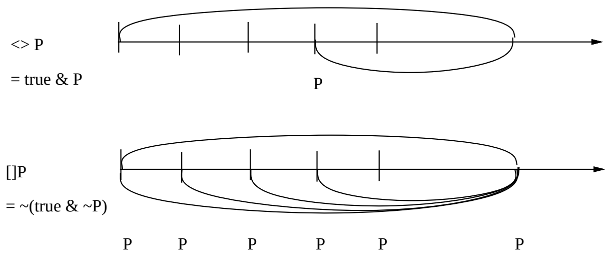
[†]Sony Computer Science Laboratory, Inc. 3-14-13 Higashi-gotanda, Shinagawa_ku, Tokyo 141, JAPAN, kono@csl.sony.co.jp



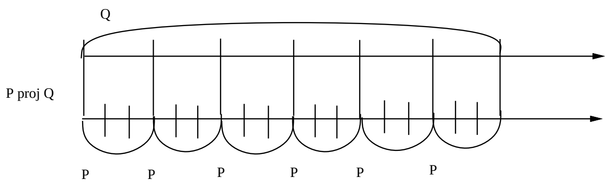
We introduce the chop operator '&' which combines two intervals. $P\&Q$ roughly means ''do P then Q''.



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



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



We shall use the following abbreviations. Here T, F means true and false.

$$\begin{aligned} \bigcirc P &\equiv @P \vee empty \\ length(n) &\equiv \underbrace{@@...@}_{n} empty \\ less(n) &\equiv \underbrace{\bigcirc\bigcirc...\bigcirc}_{n} F \end{aligned}$$

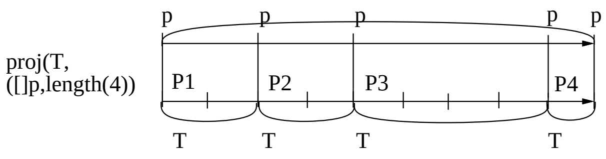
The *chop standard form* is a formula which all these abbreviations have been removed. Chop standard form may include variables and conjunction, disjunction, negation, chop, projection and existential quantifier operations.

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

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

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

Using $proj$, the time sharing task are easily described. A preemptable task p which takes 10 ticks can be represented as follows



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.

This separation is performed recursively on temporal logic operators in the formula. However, in @-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.

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. if P_e, P_i do not overlap each other,

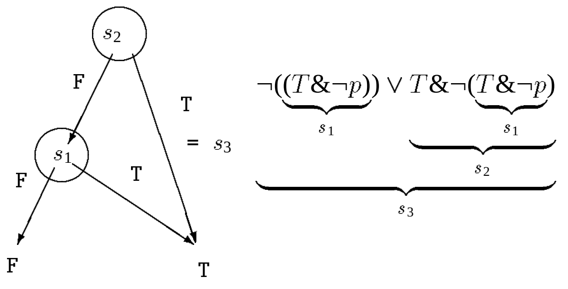
$$\vdash \neg P \Leftrightarrow (empty \wedge \neg P_e) \vee \bigvee_i P_i \wedge @\neg Px_i.$$

Our tableau expansion rules are designed to maintain this determinism. If binary subterm diagrams contain finite numbered nodes, a set of the binary subterm trees must be finite. However, we can ensure that expansion of a given ITL formula only generate finite subterms. When we expand all binary subterm diagrams, the expansion completes.

3.1 Binary Subterm Diagram

During possible world generation, various kind of ITL formulas are generated. Unlike LTTL or ETL, 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: triples ?(P, Q, R) is a binary decision node, in which if variable P is T then Q else R and numbered nodes for a binary temporal logic operator $O(P, Q)$. (ex. &, $proj$). Translation from ITL formula to binary subterm diagram is done in a bottom-up way. 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. 1), $s_3 =?(s_2, T, ?(s_1, F, T))$.



$$\underbrace{\neg(\underbrace{(T\&\neg p)}_{s_1}) \vee \underbrace{T\&\neg(\underbrace{T\&\neg p}_{s_1})}_{s_2}}_{s_3}$$

Figure 1: Binary Subterm Tree

In this way, we regard each subterm as an Binary Decision Diagram. 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.

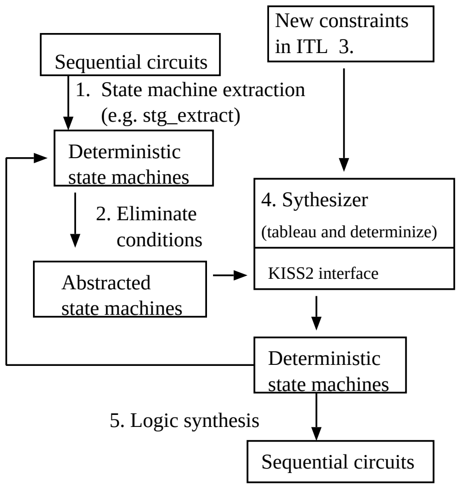

Sequential circuits
New constraints in ITL 3.
1. State machine extraction (e.g. stg_extract)
Deterministic state machines
2. Eliminate conditions
4. Sythesizer (tableau and determinize)
KISS2 interface
Abstracted state machines
Deterministic state machines
5. Logic synthesis
Sequential circuits

Figure 2: Flow of Our Redesign Method

4 A Redesign Method

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.

This redesign method for sequential circuits are shown in Figure 2. 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 2). 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 2). Then we provide ITL formulas for the specification of the eliminated variables so that they satisfy the required properties. (3. in Figure 2). By using deterministic tableau methods on the ITL formulas and the existing sequential circuits, 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 2). The final state machine can be logic synthesized and the desired sequential circuit is obtained (5. in Figure 2).

If we want a sequential circuit which satisfy the new

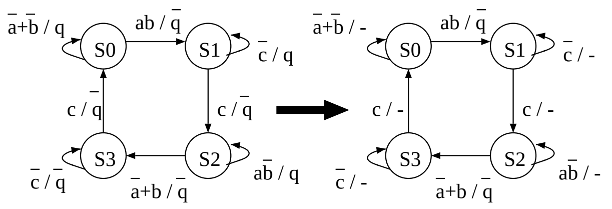


Figure 3: Modified design: some conditions for output q are eliminated

constraints but whose structure are similar to the existing circuit, we can use the method presented in [4]. 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). The synthesized state machine for $(length(2) \wedge @\Diamond p)$ $proj$ 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 3. In this circuit the ouptut 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 as in Figure 3. Here only the necessary 0 value for q is specified. If necessary, quantifiers or temporal operator can be used. Then we compute the product of the machines shown in Figure 3 and get the final state machine which satisfies the requirement for q. Resulting state machine contains 12 states and 36 transitions. From the state machine we can synthesize the final circuit using the rectification methods shown in [4]. We have tested our method on fsmexamples of SIS-1.1 from the mcnc91 distribution in KISS2 format.

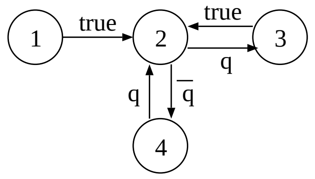


Figure 4: Synthesized state machine

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, 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.

References

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

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

[3] S. Kono. ''automatic verification of interval temporal logic''. In *Proc. 8th British Colloquium for theoretical computer science*, Mar. 1992.

[4] Y. Kukimoto and M. Fujita. ''rectification method for lookup-table type fpga's''. In *Proc. of ICCAD-92*, pp. 54--61, Nov. 1992.

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