LTL'S INTUITIVE REPRESENTATIONS AND ITS AUTOMATON TRANSLATION

Yuhong Zhao
Heinz Nixdorf Institute, University of Paderborn, Germany

Abstract: Compared with other verification methods, to some sense, model checking can be thought of as more attractive method to test hardware and software systems due to its automatic features. However, a stumbling problem is how to supply correct formal properties in logic to do model checking by system designers without specific mathematical background. This paper first presents two intuitive representations for the LTL formulas: one is graphical automaton-like; the other is textual regular-expression-like and then shows how these representations can be used to construct Büchi automata for LTL model checking.

1. INTRODUCTION

Software components have become an important part of the complex distributed (real-time) embedded systems, which usually run in a much more constrained environment than "traditional" computer systems and require consequently safety-critical and high-reliability to these systems. Therefore, one challenge today's system designers are facing is how to guarantee the correctness of such systems, especially when large concurrent and reactive systems are concerned. Moreover, in safety crucial applications, real-time requirements need to be considered, which further increase the difficulty of system development and validation. The non-determinism inherent in such applications usually makes them hard to test. However, formal methods for specifying and verifying systems can offer a greater assurance of correctness than traditional simulation and testing [CGP00].

Formal verification methods can ensure that a high-level system design really meets rigorously specified correctness requirements, thereby increase-

ing the possibility that faulty designs can be discovered at the earlier phases of system development. Temporal logics [CD88] are well-suited for specifying temporal properties of systems. Nevertheless, experiences show that specifications of even moderate-sized systems are too complex to be readily understood if without some expertise in idioms of the specification language [DAC99]. Consequently, system developers seldom make signifycant use of formal specification and verification techniques in practice.

In order to be widely adopted in the development of real world systems, formal specification and analysis methods should be made accessible to system designers and software engineers in the sense that users can express the properties of the systems about which they wish to reason as intuitively as possible and to confirm automatically that the design models of the systems satisfy the required properties. As a result, system developers can use formal specifications throughout the system lifecycle to guide development, maintenance and enhancement.

To do this, the author has presented intuitive representations for a widely used temporal logic called CTL* as well as its extensions with respect to time in [Zha03]. These representations include automaton-like graphical notations and regular-expression-like textual notations so as to fit into different needs. To some extent, these representations can offer a natural way to express system properties without sacrificing the benefits of the formal notation. Moreover, the intuitive representations of the LTL formulas can help to construct Büchi automata with features different from other methods [DGV99, Fri03, GL02, GO01, GPV+95, SB00, Tri02]. This method makes fairness constraints caused by the "U" operators disappeared and the resulting automata are the Büchi automata with only one acceptance conditions, instead of the generalized ones with multiple acceptance conditions.

Considering the limit of space, the main aim of the paper is to introduce the intuitive representations for LTL formulas and then present the automata translation method based on these intuitive representations. The remainder of this paper is structured as follows: Section 2 gives the preliminaries on linear temporal logic and on Büchi automata; Section 3 presents the intuitive representations for the LTL formulas; Section 4 addresses applying these representations to automata translation; Section 5 discusses related work and finally we draw conclusions in Section 6.

2. PRELIMINARIES

Linear Temporal Logic(LTL)[Pnu81] is composed of *temporal operators* (**X**, **F**, **G**, **U** and **R**) which specify properties of a system execution path. LTL formulas are defined inductively starting from a finite set *P* of *atomic*

propositions, the standard *Boolean* operators, and the temporal operators. Without loss of generality, given a system M, let π be an execution path; $p \in P$ be a proposition; f and g be LTL formulas. The interpretation of LTL can then be described as below:

1. $M, \pi \models p \Leftrightarrow$ p holds at the first state of π.
2. $M, \pi \models \neg f \Leftrightarrow$ f does not hold along π.
3. $M, \pi \models f \vee g \Leftrightarrow$ either f or g holds along π.
4. $M, \pi \models f \wedge g \Leftrightarrow$ both f and g hold along π.
5. $M, \pi \models \mathbf{X}f \Leftrightarrow$ f holds at the second state of π.
6. $M, \pi \models \mathbf{F}f \Leftrightarrow$ f holds at some state on π.
7. $M, \pi \models \mathbf{G}f \Leftrightarrow$ f holds at every state on π.
8. $M, \pi \models f\,\mathbf{U}\,g \Leftrightarrow$ f holds along π up to some state where g holds.
9. $M, \pi \models f\,\mathbf{R}\,g \Leftrightarrow$ g holds along π up to and including the first state where f holds.

Büchi automata are widely used in model checking to verify LTL formulas due to the characteristic that both the system model and the properties can be represented in an automaton form. There are several variants of Büchi automata. The variant typically used in model checking is Büchi automata with labels on transitions and simple accepting conditions defined in terms of states. Simply, a Büchi automata is a 6-tuple $<S, P, R, L, S_0, F>$, where S is a finite set of states, P is a finite set of propositions, $R \subseteq S \times S$ is a transition relation, $L: R \rightarrow 2^P$ is a transition labeling function, $S_0 \subseteq S$ is a set of initial states, and $F \subseteq 2^P$ is a set of accepting states.

3. LTL'S INTUITIVE REPRESENTATIONS

3.1 Graphical Representation

Without loss of generality, Figure 1 - Figure 5 illustrate the graphical representations for LTL formulas $\mathbf{X}f$, $\mathbf{F}f$, $\mathbf{G}f$, $f\,\mathbf{U}\,g$ and $f\,\mathbf{R}\,g$ respectively, each of which is composed of a dot "•" and a *path pattern.* Simply speaking, a dot "•" connects to the first position of a path pattern and a path pattern consists of *nodes* and *edges:* a node denotes a position on the path pattern, on which the formula "*f*" means only those states satisfying f can occur in this position (matching the node); an edge denotes the sequential order between states, on which a symbol "*" represents repeating zero or finitely many times and a symbol "∞" represents repeating infinitely many times. In addition, "*T*" refers to *true* representing "all states" in the system *M*; similarly, "*F*" refers to *false* representing "no state" if needed. Thus, path patterns can intuitively illustrate what states may occur in which positions on

a matching path. In this sense, a path pattern can be seen as a type of those paths matching this path pattern and such a path can be seen as an instance of this path pattern.

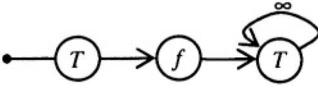

Figure 1. **X***f*

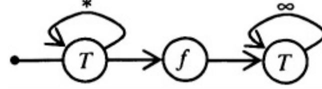

Figure 2. **F***f*

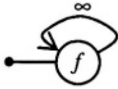

Figure 3. **G***f*

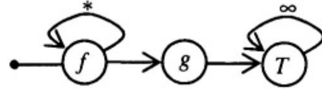

Figure 4. *f* **U** *g*

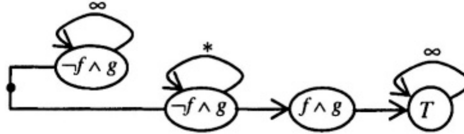

Figure 5. *f* **R** *g*

The meanings of Figure 1 - Figure 5 are obvious, i.e., each path matching the above path patterns in M are the path satisfying the corresponding formula. But how about the more complicated LTL formulas with the nested sub-formulas? Let's take the formula $\mathbf{G}(h \rightarrow \mathbf{F}(f\,\mathbf{U}\,g))$ as an example. As a result, Figure 6 is the graphical representation in which the nodes with a dot "•" characterize the nested cases. That is, a path pattern connected to the dot "•" in a node of another path pattern represents a subformula. Therefore, a path starting from a state matching such a node should, on the one hand, conform to the path pattern starting from the node and, on the other hand, conform to the path pattern of the formula in the node at the same time. For example, a path from a state matching the node N_3 should follow both the path pattern starting from N_3 and the path pattern connected to the dot "•" in N_3. Note that this is different from Figure 5 which means a path should follow one of the two given path patterns. In addition, the negation form can be represented as "¬(•)". In this way, we can intuitively represent any

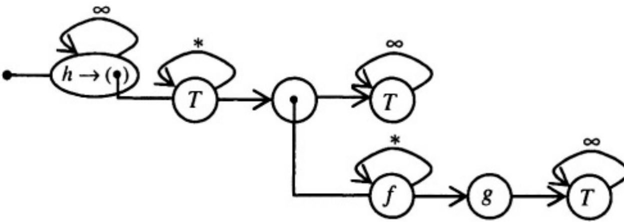

Figure 6. $\mathbf{G}(h \rightarrow \mathbf{F}(f\,\mathbf{U}\,g))$

complex LTL formulas. The *proof* is simple by structure induction and therefore omitted here.

3.2 Textual Representation

The idea of textual representation for LTL formulas is inspired by the form of regular expressions. However, in order to describe LTL formula intuitively in a textual way, we just borrow some notations of the traditional regular expressions and add some new notations to fit our needs.

The new notations related to logic operators are "~" denoting *Negation,* "|" denoting *Or,* "&" denoting *And,* "=>" denoting *Implication* and "<=>" denoting *Equivalence.* In particular, "!" is employed to force the formula immediately preceding it to repeat infinitely many times. The notations borrowed from the regular expressions are the operators related to concatenation and closure [HMU01] which also have a similar meaning here. In addition, "*T*" and "*F*" have the same meaning as in the graphical representation.

As a result, the basic LTL formula $\mathbf{X}f$ can be written as *"TfT!"*, $\mathbf{F}f$ can be written as *"T*fT!"*, $\mathbf{G}f$ written as *"f!"*, $f\,\mathbf{U}\,g$ written as *"f*gT!"*, and "$f\,\mathbf{R}\,g$" written as **"((~f&g)! | (~f&g)*(f&g)T!)"**. As for the complex LTL formulas, say $\mathbf{G}(h \rightarrow \mathbf{F}(f\,\mathbf{U}\,g))$, its regular form is also easy to be obtained in this way, i.e., "(*h* => *T**(*f***gT*!)*T*!)!". It's not difficult to reason that this textual representtation has a direct one to one mapping with the corresponding graphical representation. Therefore, its semantics is the same as the graphical one. In fact, the regular form is another way to represent path patterns. Note that, to avoid ambiguity, this regular representation has to be parenthesized whenever need. Otherwise, the meaning of the expression "(*h* => *T**(*f***gT*!)*T*!)!" would be not clear if the brackets surrounding the sub-expression "*f***gT*!" were missing.

4. LTL'S AUTOMATON TRANSLATION

Because the graphical and the regular representations for LTL formulas are essentially the same thing, here we only use the regular form to illustrate the translation procedure in this section. On the other hand, we suppose the LTL formula is of the restricted negation normal form, in which the negation is applied only to propositional variables.

4.1 Example

To ease the understanding of the translation procedure, let's first take an LTL formula $\mathbf{G}(f \rightarrow \mathbf{F}g)$ as an example. According to Section 3, the textual

form of $\mathbf{G}(f \rightarrow \mathbf{F}g)$ is $({\sim}f \mid T^*gT!)!$ representing an infinite sequence of nodes labeled with $({\sim}f \mid T^*gT!)$. If we separate the first node from the sequence, the rest of the sequence still forms an infinite sequence. That is, $({\sim}f \mid T^*gT!)!$ can be derived into two parts: $({\sim}f \mid T^*gT!)! = ({\sim}f \mid T^*gT!)::({\sim}f \mid T^*gT!)!$. Note that the double colon "::" here is employed to separate the two parts: the part before "::" is called *head;* the part after "::" is called *tail.* Intuitively, a path pattern can be seen as a sequence of nodes, in which the first node is *head* and the rest of the sequence *tail.* As for $({\sim}f \mid T^*gT!)!$, its *head* is $({\sim}f \mid T^*gT!)$ and its *tail* is $({\sim}f \mid T^*gT!)!$. However, its *head* is still a path pattern not a state formula. Our goal is to transform a path pattern into its "normal" form, i.e., its *head* is state formula. Similarly, $T^*gT! = g::T! \mid T::T^*gT!$. According to the interpretation of LTL formulas, it's not difficult to reason that

$$\begin{aligned} ({\sim}f \mid T^*gT!)! &= ({\sim}f \mid T^*gT!)::({\sim}f \mid T^*gT!)! \\ &= ({\sim}f)::({\sim}f \mid T^*gT!)! \mid (T^*gT!)::({\sim}f \mid T^*gT!)! \end{aligned}$$

Moreover, since $(T^*gT!)$ is still a path pattern, $(T^*gT!)::({\sim}f \mid T^*gT!)!$ can be further transformed as follows until all the *head* parts are state formulas:

$$\begin{aligned} (T^*gT!)::({\sim}f \mid T^*gT!)! &= (T::T^*gT! \mid g::T!)::({\sim}f \mid T^*gT!)! \\ &= T::({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!) \mid g::({\sim}f \mid T^*gT!)!\ \&\ T! \\ &= T::({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!) \mid g::({\sim}f \mid T^*gT!)! \end{aligned}$$

Notice that $T!$ matches any infinite path, so $({\sim}f \mid T^*gT!)!\ \&\ T! = ({\sim}f \mid T^*gT!)!$. $({\sim}f \mid T^*gT!)!$ can be represented as the following normal form:

$$({\sim}f \mid T^*gT!)! = ({\sim}f)::({\sim}f \mid T^*gT!)! \mid T::({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!) \mid g::({\sim}f \mid T^*gT!)!$$

Similarly, the normal form of $({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!)$ is shown as below:

$$\begin{aligned} ({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!) = \ & ({\sim}f)::({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!) \mid \\ & ({\sim}f \& g)::({\sim}f \mid T^*gT!)! \mid \\ & T::({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!) \mid \\ & g::({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!) \mid \\ & g::({\sim}f \mid T^*gT!)! \end{aligned}$$

Let $= ({\sim}f \mid T^*gT!)!$ and $B = ({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!)$, then we have

$$\begin{aligned} A &= ({\sim}f)::A \mid g::A \mid T::B = ({\sim}f \mid g)::A \mid T::B \\ B &= ({\sim}f \& g)::A \mid g::A \mid ({\sim}f)::B \mid g::B \mid T::B = g::A \mid T::B \end{aligned}$$

which can be seen as a variant of context-free grammar productions. We can construct the Büchi automaton of $\mathbf{G}(f \rightarrow \mathbf{F}g)$ as shown in Figure 7 from this production form.

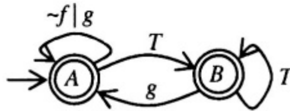

Figure 7. Büchi automaton of $\mathbf{G}(f \rightarrow \mathbf{F}g)$

One problem that needs to make further explain is accepting condition. The Büchi automaton of $\mathbf{G}(f \rightarrow \mathbf{F}g)$ contains two states A and B which are labeled with $({\sim}f \mid T^*gT!)!$ and $({\sim}f \mid T^*gT!)!\ \&\ (T^*gT!)$ respectively. Accord-

ing to the automaton construction, the state A labeled with **(~*f* | *T*****g**T*!)!** means any infinite path starting from A matches **(~*f* | *T*****g**T*!)!**; the state B labeled with **(~*f* | *T*****g**T*!)! & (*T*****g**T*!)** means any infinite path starting from B matches **(~*f* | *T*****g**T*!)!** and (*T***gT*!) at the same time. It is easy to reason that the loop from B directly to B is not acceptable because the infinite part (i.e., *T*!) of (*T***gT*!) is never matched. But the loop from B via A to B is acceptable because **(~*f* | *T*****g**T*!)! & *T*! = (~*f* | *T*****g**T*!)!**. Consequently, both A and B are accepting states with such a constraint on B that the loop from B must go through A. In general, all the states on an accepting loop can not contain a common subformula of the form "*x***yT*!". Otherwise, the loop will always matche *x***yT*! and thus can not match the infinite part of *x***yT*! at all.

Note that in this automata translation procedure we do not need fairness constraints with respect to the (implicit) **"U"** operators in the given formula. The reason is ***f* U *g* = *f*****g**T*! = *f*::*f*****g**T*! | *g*::*T*!**. Let A =*f***gT*! and B=*T*!, thus in the resulting automaton (Figure 8) only the state B labeled with *T*! is the accepting state, which guarantees that g has already been held before arriving at B. In this aspect, this automata translation method differs from many other methods by explicitly denoting the path pattern following g.

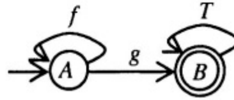

Figure 8. Büchi automaton of *f* **U** *g*

4.2 Translation Algorithm

The above example illustrates that the procedure of translating an LTL formula into Büchi automaton from its path pattern representation, which is similar to the tableau-constructing method. But the approach presented here is more simple and intuitive. Especially, the **"U"** operators are no longer a problem. In what follows, we'll present our algorithm by imitating the algorithm in [GPVW95]. Therefore, our algorithm has the same complexity as the algorithm in [GPVW95].

The basic data structure used in our algorithm is called *node*. The states of the automaton can be derived from *nodes*. A *node* is defined as below:

```
record node = [formula:   Path-Pattern,
               head:      Path-Pattern,
               tail:      Path-Pattern]
```

where the field *formula* keeps the textual path pattern of an LTL formula; the fields *head* and *tail* keep the *head* and the *tail* of the path pattern *formula* respectively. As a result, *formula* = *head::tail.* Our goal is to transform *formula* into its normal form, i.e., its *head* is a state formula. Obviously, the

three fields together can uniquely identify a *node*. Intuitively, the successors of a *node* are the *nodes* with *formulas* the same as *tail* of the *node* and the predecessors of a *node* are the *nodes* with *tails* the same as *formula* of the *node*. The edge between a *node* and its successor is labeled with *head* of the *node*. To some sense, the resulting automaton is a flattened path pattern.

Given an LTL formula *r* in a textual path pattern form, the function *create_graph* initiates the automaton construction procedure by applying *split* to the starting node with *formula* and *head* set to *r* and *tail* set to *T*!. As a result, *create_graph* returns a set of nodes, from which we can derive a Büchi automaton of *r*.

```
function create_graph(r: Path-Pattern): Node-Set
  return (split([formula ⇐ r, head ⇐ r, tail ⇐ T!], ∅));
end function
```

The recursive function *split* builds a tableau. It has two parameters: *node* denoting the current *node* to be processed and *node_set* a set of *nodes* have been generated by now and returns an updated set of *nodes* if possible.

```
function split(node: Node, node_set: Node-Set): Node-Set
  if ∃node'∈ node_set: head(node) = formula(node') then
     for each node'∈ node_set: head(node) = formula(node')
        node_set := node_set ∪ [formula   ⇐ formula(node),
                                head      ⇐ head(node'),
                                tail      ⇐ tail(node) & tail(node')];
     end for
     return (node_set);
  else if head(node) is a state formula then
     if head(node) ≠ F then
        return(split([formula ⇐ tail(node), head ⇐ tail(node), tail ⇐ T!],
                    node_Set ∪ {node}));
     else return (node_set);
     end if
  else if head(node) is of form f | g then
     node_set := split([formula ⇐ formula(node), head ⇐ f, tail ⇐ tail(node)],
                      node_set);
     return (split([formula ⇐ formula(node), head ⇐ g, tail ⇐ tail(node)], node_set));
  else if head(node) is of form f & g then
     node_set := split([formula ⇐ f, head ⇐ f, tail ⇐ T!], node_set);
     node_set := split([formula ⇐ g, head ⇐ g, tail ⇐ T!], node_set);
     for each node1 in node_set with formula(node1) = f and
         each node2 in node_set with formula(node2) = g do
        node_set := split([formula ⇐ formula(node),
                          head    ⇐ head(node1) & head(node2),
                          tail   ⇐ tail(node) & tail(node1) & tail(node2)], node_set);
     end for
```

```
            return(node_set);
        else if head(node) is of form TfT! then
            return(split([formula ⇐ formula(node),
                           head    ⇐ T,
                           tail    ⇐ tail(node) & fT!], node_set));
        else if head(node) is of form f*gT! then
            node_set := split([formula ⇐ formula(node),
                               head    ⇐ g,
                               tail    ⇐ tail(Node) & T!], node_set);
            return(split([formula ⇐ formula(node),
                           head    ⇐ f,
                           tail    ⇐ tail(Node) & f*gT!], node_set));
        else if head(node) is of form f! then
            return(split([formula ⇐ formula(node), head ⇐ f, tail ⇐ tail(node) & f!],
                    node_set));
        end if
    end function
```

4.3 Acceptance Condition

We can deduce the accepting states of the resulting automaton as follows. According to our automaton construction, a state labeled with a path pattern means any infinite path (loop) starting from the state matches the path pattern. Consequently, the infinite part of the path matches the infinite part of the path pattern. In addition, the *tails* of the *nodes* obtained from *create_graph* always have the conjunction form "x_1 & x_2 & … & x_n" ($n \geq 1$) where x_i has a form of either "x*yT!" or "x!". Notice that a path pattern of the form "TxT!" or "xT!" is a special case of the form "x*yT!". This conjunction form requires any loop from the corresponding state in the automaton match the n path patterns $x_1, x_2, \ldots, x_n$ at the same time.

For convenience, we denote "x_1 & x_2 & … & x_n" as a path pattern set $\{x_1, x_2, \ldots, x_n\}$. If all the states $s_1, s_2, \ldots, s_k$ on a loop share common path patterns of the form "x*yT!", say, f*$gT! \in path_pattern_set(s_1) \cap path_pattern_set(s_2) \cap \ldots \cap path_pattern_set(s_k)$), obviously, the loop will match f*gT! forever but never get to the infinite part *of* f*gT!. Consequently, such a loop is not acceptable. However, if the above condition is not *true,* then we can say the loop is acceptable. In general, if a state in a resulting automaton has a loop starting from it and the states on the loop do not share common path patterns of the form "x*yT!", then the state is a (constrained) accepting state.

It's easy to reason that a state labeled with a path pattern of the form "y_1! & y_2! & … & y_n!" is definitely an accepting state and a state labeled with a path pattern of the form "x_1*y_1T! & x_2*y_2T! & … & x_n*y_nT!" is definitely not an accepting state. Therefore, a state contains path patterns of the two forms

"*x*!" and "*x***yT*!" is an accepting state if there is a loop to it and the states on the loop do not share common path patterns of the form "*x***yT*!".

As mentioned in section 4.1, this automata translation procedure does not need fairness constraints with respect to the (implicit) "**U**" operators in the given formula and thereby differs from many other methods. According to the existing methods, for each subformula of the form "f **U** g", a set of accepting states $F_{f \mathbf{U} g} = \{s \in S: f \mathbf{U} g \notin s \text{ or } g \in s\}$ is produced. In case a formula contains multiple subformulas of the form "f **U** g", then the resulting automaton contains accordingly multiple sets of accepting states, so called generalized Büchi automaton. The path of a generalized Büchi automaton is accepted if for each set F_i of accepting states, there are infinitely many s's on the path such that $s \in F_i$. Therefore, a generalized Büchi automaton is usually transformed into a normal Büchi automaton with only one set of accepting states by using a counter i: each state becomes a pair $<s, i>$. The counter is initialized to 0 and counts modulo m ($m = |F|$) where $F = \{F_0, F_1, \ldots, F_{m-1}\}$. It is increased whenever a state of the ith set $F_i \subseteq F$ is reached. As a result, only one set of accepting states, say $F_0 \times \{0\}$, is needed.

5. RELATED WORK

Graphical representation, due to its visual effect, is popular in the process of system development. Some intuitive representations for temporal logic properties have been presented in recent years. Timing Diagrams [SD93] are a graphical notation for expressing precedence and causality relationships between events in a computation, the semantics of which is defined by a subset of temporal logics. Graphical Interval Logic (GIL) [DKM+94] is a visual temporal logic in which formulas resemble timing diagrams and can thus express a subset of temporal logic, too. Timeline notation captures the event-based LTL requirements [SHE01]. Constrained expression representation in [ABC+91] is essentially a regular expression which can not address infinite executions of the system. Regular CTL (RCTL) [BBL98] covers a rich and useful set of CTL formulas and regular expressions. Bandera Specification Language (BSL) [CDH+00] is a source-level model checking independent language for expressing properties of Java program actions and data.

For the automaton translation, many existing approaches [DGV99, Fri03, GL02, GO01, GPV+95, SB00, Tri02] are mainly based on the LTL formulas or alternating Büchi automata of the LTL formulas together with some simplification and optimization techniques to reduce the size of the resulting automata. In contrast, the method presented in this paper built the automata based on the path patterns of the LTL formulas. Path pattern is similar but different from alternating Büchi automaton in concept and use. We just use

path pattern (formula) "x_1 & x_2 & ... & x_n" to label a state, but never think of such a state as a conjunction of n basic states labeled respectively with x_1, x_2, ..., x_n. A path pattern can characterize the whole path instead of the prefix of the path, say **F**g, because path pattern ends with the infinite form "x!". Thus, path pattern does not need accepting states. Using path pattern does not need to transform **G**(**F**) into **R**(**U**) operator. Checking a (constrained) accepting state is simple and the resulting automaton can directly be used to do LTL model checking. That is to say, we can avoid the problem caused by the formula of form "f **U** g" and obtain a "normal" Büchi automaton directly.

6. CONCLUSION AND OUTLOOK

Expressing complex requirements in logic is without doubt a challenging task. Therefore, this paper attempts to visualize the cryptic specifications to ease the question. By using path pattern, one can intuitively reason what type of states can occur in which positions on a path and both state- and event-based properties can be specified in a unified way. Moreover, path pattern can help to construct the normal Büchi automata, instead of the generalized ones, which different from many other translation methods in the aspect that this method avoids the problem caused by the **"U"** operator naturally. We plan to study on the simplification and optimization methods related to this automata translation way in the future.

REFERENCES

[ABC+91] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated Analysis of Concurrent Systems with the Constrained Expression Tool-set. IEEE Transactions on Software Engineering, 17(11): 1024-1222, Nov. 1991.

[BBL98] I. Beer, S. Ben-David, A. Landver. On-the-fly Model Checking of RCTL Formulas. CAV'98, LNCS 1427, pp. 184-194.

[CD88] E. M. Clark, I. A. Draghicescu. Expressibility results for linear time and branching time logics. In Linear time, Branching time, and Partial order in Logics and Models for Concurrency, LNCS 354, pp. 428-437. Springer, 1988.

[CDH+00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A Language Framework for Expressing Checkable Properties of Dynamic Software. In SPIN Software Model Checking Workshop, pp. 205-223. Stanford, CA. 2000.

[CGP00] E. M. Clark, Jr., O. Grumberg, and D. A. Peled. Model Checking. MIT Press. 1999.

[DAC99] M. B. Dwyer, G. S. Avrunin and J. C. Corbett. Pat-terns in Property Specifications for Finite-State Verification. In Proc. of the 21st International Conf. on Software Engineering, pp. 411-420. May, 1999.

[DGV99] M. Daniele, F. Giunchiglia, and M. Vardi. Improved Automata Generation for Linear Temporal Logic. In Proc. of the 11th International Conference on Computer Aided Verification (CAV'99), Trento, Italy. Springer, LNCS1631.

[DKM+94] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A Graphical Interval Logic for Specifying Concurrent Systems. ACM Transactions on Software Engineering and Methodology, 3(2): 131-165, Apr. 1994.

[Fri03] C. Fritz. Constructing Büchi Automata from Linear Temporal Logic Using Simulation Relations for Alternating Büchi Automata. In CIAA 2003, LNCS 2759, pp. 35-48, 2003.

[GL02] D. Giannakopoulou and F. Lerda. From States to Transitions: Improving Translation of LTL formulae to Büchi Automata. In Formal Techniques for Networked and Distributed Systems - FORTE 2002, LNCS 2529, pp. 308-326, Texas, USA, November, 2002.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In Proc. of the 13th International Conference on Computer Aided Verification (CAV'01). July, 2001, Paris, France. Springer, LNCS 2102.

[GPV+95] R.Gerth, D. Peled, M. Vardi and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95). June, 1995, Warsaw, Poland.

[GPVW95] R.Gerth, D. Peled, M. Vardi and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In Proc. of the 15th IFIP/WG6.1 Symposium on Protocol Specification, Testing and Verification (PSTV'95). June, 1995, Warsaw, Poland.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to Automata Theory, Language, and Computation (second edition). Addison-Wesley, 2001.

[Pnu81] A. Pnueli. A temporal logic of concurrent programs. Theoretical Computer Science 13: 45-60.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In Computer Aided Verification, 12th International Conference (CAV2000), LNCS 1855, pp. 249-263, 2000.

[SD93] R. Schlör and W. Damm. Specification of system-level hardware designs using timing diagrams. In Proc. Europe Conf. Design Automation and Europe Event in ASIC Design, pages 518-524, Paris, Feb. 1993. IEEE Computer Society Press.

[SHE01] M.H. Smith, G. J. Holzmann and K. Etessami. Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs. In the 5th IEEE International Symposium on Requirements Engineering, pp. 14-23. Canada, August, 2001.

[Tri02] X. Thirioux. Simple and Efficient Translation from LTL Formulas to Büchi Automata. In Electronic Notes in Theoretical Computer Science 66 No. 2(2002).

[Zha03] Y. Zhao. Intuitive Representations for Temporal Logic Formulas. In Proc. of Forum on Specification and Design Language (FDL'03), pp. 405-413, Frankfurt, Germany, September, 2003.