Atlantic Electronic
Journal of Mathematics
Volume **1**, Number **1**, Summer **2006**

http://aejm.ca
http://rema.ca
pp. **69–86**

AUTOMATED RECOGNITION OF STUTTER INVARIANCE OF LTL FORMULAS

Jeffrey Dallien[1] and Wendy MacCaull[2]

Department of Mathematics, Statistics and Computer Science
St. Francis Xavier University
Antigonish, N.S. B2G 2W5

Abstract. Formal verification is the growing field of formalizing and verifying specifications for hardware and software systems. One such technique, known as model checking, can exhaustively check a software model for properties written in a specification language, such as Linear Temporal Logic (LTL). For large systems, memory reduction techniques are often used to complete a verification with the available hardware. One reduction technique, that of partial order reduction, has proven useful in aiding verifications, but it limits the expressiveness of LTL specifications. Due to the phenomenon of stuttered states in the model's reduced state graph representation, the LTL next operator is not guaranteed to produce correct results when partial order reduction is used. As more casual users begin using model checking, it is important to provide them with a convenient method of writing specifications that are known to be safe to use in the presence of partial order reduction.

A description of how stuttered states are introduced and of a system of patterns for LTL formulas that are safe to use, despite the fact that they use the next operator, are given. To automatically determine if a formula matches one of the safe patterns, a Prolog based LTL recognizer which we developed is discussed. A listing of the patterns of formulas recognized is given as an appendix.

1. Introduction.

1.1. **Model Checking.** Formal verification is a growing field aimed at formalizing specifications for software and hardware systems. This is accomplished using methods to automatically verify that these systems satisfy their specifications. The ultimate goal of formal verification is to exhaustively verify that every behaviour of a software or hardware system is in accordance with the intended behaviours of the programmers or engineers designing the system.

One method of formal verification, known as *model checking*, uses an abstract model of the system to check if a property holds. Automated tools, known as *model checkers* are used to generate all execution paths (behaviours) in the modeled system, in order to verify that all reachable paths comply with the properties specified.

2000 *Mathematics Subject Classification.* Primary: 68Q60, 68Q45; Secondary: 03B44, 68Q85.

Key words and phrases. formal methods, verification, model checking, Linear Temporal Logic, automated pattern matching.

[1]Support from an NSERC Research Capacity Development Graduate Fellowship is gratefully acknowledged.

[2]Support from Science and Engineering Research Council of Canada is gratefully acknowledged.

Counter-examples are provided for a property if a behaviour violates it, allowing the system's operation to be corrected.

Many model checkers have been developed, employing different combinations of strategies and features for the purpose of verifying hardware and software. One such model checker for Linear Temporal Logic (LTL), known as Spin, has been developed and maintained by Gerard Holzmann at Bell Labs over the past two decades [5, 6]. Originally designed to verify communication protocols, its ability to model and verify properties on asynchronous processes has resulted in it being successfully applied to a wide range of systems.

1.2. **Temporal Logics.** Often when specifying properties of software systems, it is necessary to express some concept of time, such as that one condition becomes true after another, or some condition will always become true in the future. Propositional and predicate logic cannot express such relationships over time; temporal logics were developed to provide a means of expressing these types of properties.

One such logic is Linear Temporal Logic (LTL). LTL is used to specify properties pertaining to an execution path of a model. LTL model checker software will generate all the execution paths of a model, checking that a given LTL property holds on each. The Spin model checker accepts model properties written in LTL.

Linear Temporal Logic provides four temporal operators for use in logic expressions: $\Box$, *always*; $\Diamond$, *eventually*; $\mathcal{U}$, *until*; and $\mathcal{X}$, *next*. This paper focuses on the infrequently used next operator. The reasons for its lack of use with Spin and a description of when LTL formulas are safe to use with Spin are discussed. As more casual users begin using model checking tools, it is important to provide a simple means of identifying formulas which are safe to use, even for those users who are not very familiar with logic. With this goal in mind, we implement a recognizer to determine if a given formula matches one known to be safe. More details of the results reported here may be found in [2].

2. Basic Definitions and Theorems.

Before describing concepts and theorems in this paper, some basic definitions are necessary. More complex and specific definitions will be provided in the relevant sections of this paper, as they are needed.

2.1. Model and LTL Definitions.

Definition 1. Let AP be a set of atomic propositions. A Kripke structure, $\mathcal{M}$ over AP is a four tuple, $\mathcal{M} = (S, S_0, R, L)$ [1], where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
4. $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

Models of systems may be represented by Kripke structures and are often referred to as Kripke models. The description of a state consists of the values of the variables in the system; a change in a value means change in state. The set of initial states of $\mathcal{M}$ indentifies the states from which processing can begin. A transition is a pair (s, s') from the relation R indicating a possible change in state from s to s'.

FIGURE 1. An example Kripke model, $\mathcal{M}_1$

A *path*, p, in the structure $\mathcal{M}$ from a state s is an infinite sequence p of states $s_0, s_1, s_2, ...$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$. We denote by p_i the suffix $s_i, s_{i+1}...$ of the path p.

2.2. **LTL Syntax and Semantics.** We first give the syntax of LTL in BNF form:

$$\phi := a \mid \neg\phi \mid \phi \rightarrow \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \Box\phi \mid \Diamond\phi \mid \mathcal{X}\phi \mid \phi\mathcal{U}\phi$$

where a is a propositional variable. As usual, $\phi \rightarrow \psi$ is shorthand for $\neg\phi \vee \psi$ and $\phi \vee \psi$ is shorthand for $\neg(\neg\phi \wedge \neg\psi)$. The unary operators, $\neg, \Box, \Diamond, \mathcal{X}$, bind most closely, followed by $\wedge, \vee, \rightarrow$ and $\mathcal{U}$. The subset of LTL formulas not containing the $\mathcal{X}$ operator is denoted as LTL_x.

Next, the semantics of LTL are defined with respect to a Kripke model. Let $\mathcal{M}$ be a Kripke model, let $p = s_0, s_1...$ be a path in the model $\mathcal{M}$, let F_1 and F_2 be LTL formulas, and let a be a propositional variable. The notation $\mathcal{M}, p \vDash F_1$ will be used to mean that formula F_1 holds or is satisfied along the path p in the model $\mathcal{M}$. The satisfaction relation, $\vDash$, is formally defined inductively as follows:

$$\begin{array}{lcl}
\mathcal{M}, p \vDash a & \iff & a \in L(s_0) \\
\mathcal{M}, p \vDash \neg F & \iff & \mathcal{M}, p \nvDash F \\
\mathcal{M}, p \vDash F_1 \wedge F_2 & \iff & \mathcal{M}, p \vDash F_1 \text{ and } \mathcal{M}, p \vDash F_2 \\
\mathcal{M}, p \vDash \mathcal{X}F_1 & \iff & \mathcal{M}, p_1 \vDash F_1 \\
\mathcal{M}, p \vDash \Box F_1 & \iff & \forall i \geq 0 \cdot \mathcal{M}, p_i \vDash F_1 \\
\mathcal{M}, p \vDash \Diamond F_1 & \iff & \exists i \geq 0 \cdot \mathcal{M}, p_i \vDash F_1 \\
\mathcal{M}, p \vDash F_1 \mathcal{U} F_2 & \iff & \exists k \geq 0 \cdot \mathcal{M}, p_k \vDash F_2 \text{ and} \\
 & & \quad \forall j \ 0 \leq j < k \cdot \mathcal{M}, p_j \vDash F_1
\end{array}$$

Some basic equivalences of LTL operators are:

$$\neg\Box F \equiv \Diamond\neg F \tag{2.1}$$

$$\neg\Diamond F \equiv \Box\neg F \tag{2.2}$$

$$\Box F \equiv \neg(\top \ \mathcal{U} \ \neg F) \tag{2.3}$$

$$\Diamond F \equiv \top \ \mathcal{U} \ F \tag{2.4}$$

$$\neg\mathcal{X}F \equiv \mathcal{X}\neg F \tag{2.5}$$

We say that $\mathcal{M}, s \vDash F$ iff $\mathcal{M}, p \vDash F$ for every path p with s as its first state. An LTL Formula F is true in a model $\mathcal{M}$, written $\mathcal{M} \vDash F$, iff for all paths p starting at initial points $\mathcal{M}, p \vDash F$.

Example 2.1. To illustrate some of the concepts of Kripke models and LTL, an example Kripke model, $\mathcal{M}_1$, is given in Figure 1. Each state is represented by a circle and the letters inside each circle indicate which propositional variables are true at that state. The transitions, defined by the R relation, are shown as arrows between states. There is one initial state, s_0, and two possible paths, $p_1 = s_0, s_1, s_2$ and $p_2 = s_0, s_1, s_3$, in this model.

The model $\mathcal{M}_1$ and path p_1 satisfy the LTL formula $\Box x$, read "always x", because the variable x is true in every state of the path. The model $\mathcal{M}_1$ and the initial state s_0 satisfy the LTL formula $\Diamond y$, read "eventually y", because y eventually becomes true along both paths beginning from s_0. The model $\mathcal{M}_1$ and the initial state s_0 do not satisfy the LTL formula $x \ \mathcal{U} \ z$ because x must remain true until z becomes true but this does not happen on path p_2.

FIGURE 2. Two stutter equivalent paths

2.3. **Stuttering and Stutter Equivalence.**

Definition 2. Stuttering occurs in a path p of states when a state occurs two or more times consecutively; that is, for $p = s_0, s_1, s_2, ...$ if there is some $i \in \mathbb{N}$ such that $s_i = s_{i+1}$ so that $p = s_0, s_1, ...s_i, s_i, s_{i+2}, ...$

Definition 3. Two infinite paths $p = s_0, s_1, ...$ and $p' = r_0, r_1, ...$ are stutter equivalent, written as $p \sim_{st} p'$, if there are two infinite sequences of positive integers $0 = i_0 < i_1 < i_2 < ...$ and $0 = j_0 < j_1 < j_2 < ...$ such that for every $k \geq 0$,

$$L(s_{i_k}) = L(s_{i_k+1}) = ... = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1})... = L(r_{j_{k+1}-1}).$$

Figure 2 will help the reader understand this definition. Two paths are stutter equivalent if they can be partitioned into blocks such that the states in the k^{th} block are labeled the same in both paths. Note that the blocks can be of different sizes.

Stuttered states can occur normally and without problem when loops occur in a system's Kripke model. Problems can arise from apparent stuttered states caused by variable abstraction, which will be detailed further in the next section.

The definition of stutter equivalence can be extended to structures. The structures M and M' are stutter equivalent if and only if:

- M and M' have the same set of initial states;
- for each path p of M that starts from an initial state s of M, there exists a path p' of M' from the same initial state s such that $p \sim_{st} p'$;
- for each path p' of M' that starts from an initial state s of M', there exists a path p of M from the same initial state s such that $p' \sim_{st} p$.

Corollary 1. *Let M and M' be two stutter equivalent structures. Then, for every LTL_x property F, and every initial state $s \in S_0, M, s \vDash F$ if and only if $M', s \vDash F$.*

We denote the set of natural numbers as $\mathbb{N}$; and use $\neg$, $\wedge$, $\vee$ ($\bigwedge, \bigvee$) as boolean (infinitary boolean) operations.

Definition 4. The interpretation of an LTL formula F on a path p denoted as $[\![F]\!]_p$, is its truth value on that path; ie. $[\![F]\!]_p = \top$ iff $p \models F$. From the definition of satisfaction we see that for all LTL formulas A,

$$\begin{aligned}
[\![a]\!]_p &= \top \text{ if } a \in L(s_0) \\
[\![\neg A]\!]_p &= \neg [\![A]\!]_p \\
[\![A \wedge B]\!]_p &= [\![A]\!]_p \wedge [\![B]\!]_p \\
[\![\mathcal{X} A]\!]_p &= [\![A]\!]_{p_1} \\
[\![\Box A]\!]_p &= \bigwedge i \in \mathbb{N} \cdot [\![A]\!]_{p_i} \\
[\![\Diamond A]\!]_p &= \bigvee i \in \mathbb{N} \cdot [\![A]\!]_{p_i} \\
[\![A \mathcal{U} B]\!]_p &= \bigvee i \in \mathbb{N} \cdot \left([\![B]\!]_{p_i} \wedge \left(\bigwedge_{j=0}^{i-1} [\![A]\!]_{p_j} \right) \right)
\end{aligned}$$

3. **Partial Order Reduction.** Model checking of systems with asynchronous processes is very difficult. In order to verify that all possible behaviours of the system do not violate a property, all the possible combinations of states and transitions over all the asynchronous processes must be considered. Interleaving is a way to represent the result of asynchronous processes because any of a variety of sequences of states may result in any execution. The combination of states and transitions creates an interleaving of the asynchronous state graphs into a single larger state graph.

In an effort to reduce the number of paths explored and the number of states that must be stored during verification, a technique known as *partial order reduction*, first developed by Holzmann and Peled [7], is often used. The algorithm takes its name from its early versions that were based on the partial order model of program execution [1]. This reduction technique takes advantage of the fact that for a given property, the order in which states occur in a given interleaving sequence may have the same result as a different sequence of the same states. Partial order reduction groups sequences in the set of all interleaved sequences into equivalence classes: sequences of states that have the same effect for the given property are grouped together. Only one representative for a given equivalence class needs to be checked and stored, greatly reducing the number of paths and states in memory, which then allows larger models to be verified. The reduction algorithm used by Holzmann and Peled in [7] can identify instances where ignoring an interleaving of states will not result in a different verification result [1].

We may view the system states as consisting of only the variables which affect a property being checked. As we detail in the following paragraph, partial order reduction can provide a state graph that has paths which are stutter equivalent to the paths of the full state graph in terms of this property. Since a state is simply a listing of the values of the variables, ignoring some variables can produce execution paths where some state stutters. Some states may stutter more times in the reduced graph than in the full state graph, but there will be fewer sequences to check.

An example state graph will be used to illustrate how the partial order reduction algorithm in [7] introduces stuttering in its reduced state graphs. A transition is considered *invisible* if it does not change the values of the propositional variables in a correctness property.

Consider a correctness claim based on the variables x and y on the state graph in Figure 3. All transitions labelled a are invisible, that is, they do not change the value of either x or y. The transitions labelled b_1, b_2, b_3 do change the variables x and y and are therefore not invisible. The state graph shows many possible execution sequences, though many are equivalent.

An example of one of the many paths in the unreduced state graph is seen in Figure 4. By executing the transitions based on the rules of the partial order reduction algorithm, we get the state graph with a single path seen in Figure 5.

In terms of the claim being verified, which relates solely to the x and y variables, the order of unique states is the same. However, the reduced state graph stutters a different state than the unreduced path depending on when the invisible transition, a, is executed. As indicated by the dashed vertical lines, the two paths can be partitioned into blocks such that the kth block contains states with the same labels in both paths, as per the definition of stutter equivalence.

FIGURE 3. A reducible state graph with invisible transitions

The advantage is that instead of checking all four possible execution paths in the full state graph, the model checker will only need to consider a single path by using the reduced graph. If any of the paths satisfy the claim, they all do.

Controlling state space explosion is a primary concern in most verification tasks being performed today. In practice, the partial order reduction algorithm has been shown to significantly decrease the size of the state space generated when verifying properties in the best cases, and to not significantly increase the problem in the worst cases [7].

The reduction has proved useful enough to be implemented in a widely used model checker, Spin [5, 6], and its use there has taken precedence over the use of the LTL next operator in specifying correctness properties.

4. **Stutter Invariance and LTL Patterns.**

Formulas involving the next operator are problematic if partial order reduction is used, due to stuttering. Also, documentation for the Spin model checker implies that stuttering can be introduced when the model source code is converted to intermediate code [4].

Stuttering has no adverse effects on formulas containing any of the LTL operators except for next, because the other temporal operators, $\Box$ (always), $\Diamond$ (eventually), and $\mathcal{U}$ (until), do not imply any specific number of states to which their temporal properties apply. The $\Box$ (always) operator indicates that in every state some proposition must be true: a repeated state will not affect the truth value of a formula quantified with always. The $\Diamond$ (eventually) operator indicates that at some point in the future some proposition must hold. Any number of states, including repeated states, can occur first without affecting the interpretation of the formula. The $\mathcal{U}$ (until) operator requires that one proposition hold until another one does. A repeated state could delay the second proposition of an until formula from becoming true, but could not change whether it eventually becomes true or not. Clearly then, formulas using only these operators are stutter invariant because they are not affected by repeating states.

Conversely, the $\mathcal{X}$ (next) operator indicates that in whatever state occurs after a single transition, some proposition must hold. If a state is stuttered after partial order reduction (or otherwise), the next operator could be evaluated on the repeated state and not the actual next state in the system. Thus not all formulas using the next operator are safe to use, since the use of formulas affected by stuttering could lead to incorrect verification.

The benefits of partial order reduction in saving states, combined with the uncertainty of the possible introduction of stuttering by the model checker, have led to the general sentiment that the next operator should not be used at all when writing LTL properties for model checking. This is unfortunate because the next operator can be used for specifying many useful properties, such as those involving events [10].

We wish to have the convenience of expressing LTL properties using the next operator, without the risk of misinterpretation that stuttering introduces. This leads us to the following definition.

4.1. **Stutter Invariance.**

FIGURE 4. An example path from the unreduced state graph

FIGURE 5. Reduced state graph after partial order reduction

Definition 5. An LTL formula F is stutter invariant with respect to a Kripke structure $\mathcal{M}$ if and only if for each pair of paths p and p', such that $p \sim_{st} p'$,

$$p \vDash F \text{ if and only if } p' \vDash F.$$

The set of stutter invariant LTL formulas will be denoted by I.

Proposition 1. *The following are some closure properties of I:*

$$\begin{aligned}
\textit{If } a \textit{ is a variable} \quad & \textit{then} \quad a \in I && (4.1)\\
\textit{If } A \in I \quad & \textit{then} \quad \neg A \in I && (4.2)\\
\textit{If } A \in I \textit{ and } B \in I \quad & \textit{then} \quad (A \wedge B) \in I && (4.3)\\
\textit{If } A \in I \quad & \textit{then} \quad \Box A \in I && (4.4)\\
\textit{If } A \in I \quad & \textit{then} \quad \Diamond A \in I && (4.5)\\
\textit{If } A \in I \textit{ and } B \in I \quad & \textit{then} \quad (A \mathcal{U} B) \in I && (4.6)
\end{aligned}$$

Proof. If a is a variable then $a \in I$. By the semantics of LTL, a path $p = s_0, s_1, ...$ satisfies a propositional variable, a, if $a \in L(s_0)$. Only the first state is relevant in the satisfaction of a, so subsequent states, stuttered or not, will have no effect on the satisfaction of a on that path, or equivalently, will not change the interpretation of a on that path. Thus, a is stutter invariant.

The other properties can be shown similarly; see [9] or [2]. □

Remark. 4.6 is espeically important because with the exception of $\mathcal{X}$, all other LTL temporal operators can be expressed in terms of $\mathcal{U}$.

Corollary 2.

If $A \in I$ and $B \in I$ then $(A * B) \in I$ where $*$ is any of $\wedge, \vee, \rightarrow, \leftarrow$ or $\leftrightarrow$.

Proof. As seen in (4.2) and (4.3), stutter invariance is closed under negation and conjunction. The result follows because conjunction and negation form an adequate set for Boolean connectives. □

Corollary 3.

$$\textit{If } A \in LTL\_x \textit{ then } A \in I.$$

Stutter invariant formulas are not subject to misinterpretation due to stuttering and thus are safe to use with partial order reduction and with Spin's intermediate code. It has been shown that if a formula containing the next operator is stutter invariant then it may be translated to a formula without next to which it is equivalent in meaning [11]. However, an efficient means of performing this translation is not available for any given LTL formula. Also, the translation process may add states to the state graph generated from the formula, which in turn makes verification more difficult, something we are trying to avoid. In addition, as model checking tools become more popular, more casual users who are less familiar with the inner workings of the tools will start to use them. An automated way of detecting stutter

invariance for LTL formulas will be much easier for these users than requiring a complex translation to another formula with an equivalent meaning.

Some formulas that contain the next operator are stutter invariant. Determining whether a formula is stutter invariant for a Kripke structure that accepts infinite sequences of states has been shown to be PSPACE-complete [12]. However, patterns of useful formulas which are stutter invariant have been determined [10]. Currently Spin does not implement any method of checking for stutter invariance; when running the model checker, only a warning is printed that stutter invariance is required for LTL properties when using partial order reduction. First a discussion on how the next operator can be used safely by using these patterns is given, then an LTL recognizer which will attempt to match a given LTL formula with a known stutter invariant pattern will be presented to help make the next operator easier to use when specifying properties for model checking.

4.2. **Edges.** To create formulas that are stutter invariant even with the next operator, the concept of edges is used [10]. An edge in temporal logic refers to a change in a variable, which guarantees that a change in state has occurred and that the same state has not simply been repeated.

For example, consider the LTL formula $\neg A \wedge \mathcal{X}A$. This says that the variable A is not true in the current state but is true in the next state. If A represented whether a robotic arm is holding an item, $\neg A \wedge \mathcal{X}A$ would represent going from not holding to holding, or the event of picking up the item. Similarly, $A \wedge \mathcal{X}\neg A$ would represent the event of releasing the item. This change in value involves two consecutive transitions in the system and is called an *edge*.

Definition 6. The following notation [10] will be used for edges:

$$\uparrow A = \neg A \wedge \mathcal{X}A \qquad \text{up or rising edge}$$
$$\downarrow A = A \wedge \mathcal{X}\neg A \qquad \text{down or falling edge}$$

A rising edge occurs when the variable is false in one state and true in the next. Conversely, a falling edge occurs when the variable is true in one state and false in the next.

4.3. **Properties of Edges and Stutter Invariant Formulas.** Consider the formula $\mathcal{X}B$ where $B \in I$. This formula is not stutter invariant: if $s \vDash \mathcal{X}B$ and if the first state of s is stuttered to produce s', then possibly $s' \nvDash \mathcal{X}B$. Stuttering any state other than the first does not affect the interpretation of $\mathcal{X}B$.

To create a stutter invariant formula from $\mathcal{X}B$, we may add two things: the first is an edge of a stutter invariant formula, A. We now have a new formula, $\uparrow A \wedge \mathcal{X}B$, or, $\neg A \wedge \mathcal{X}A \wedge \mathcal{X}B$. The second is the temporal quantifier, $\Diamond$. The quantifier is needed to say that "eventually" our formula will be true, allowing the interpretation of the formula to apply after any stuttered initial state. The resulting formula, $\Diamond(\neg A \wedge \mathcal{X}A \wedge \mathcal{X}B)$ is stutter invariant; any repeated state in a path will not affect the interpretation of this formula. We have proved the following:

Theorem 1.

$$\textit{If } A \in I \textit{ and } B \in I \textit{ then } \Diamond(\neg A \wedge \mathcal{X}A \wedge \mathcal{X}B) \in I.$$

This theorem forms the basis for using edges in more complex formulas that are also stutter invariant. One such complex formula is found in the following theorem.

Theorem 2.

$$\textit{If } A, B, C, D, E, F \in I \textit{ then } (\neg \uparrow A \vee \mathcal{X}B \vee C)\mathcal{U}(\uparrow D \wedge \mathcal{X}E \wedge F) \in I.$$

Proof. The proof of this theorem is based on the fact that $A\mathcal{U}B$ is stutter invariant when A and B are, by Proposition 1. It remains to show that if $A, B, C, D, E, F \in I$, then $\neg \uparrow A \vee \mathcal{X}B \vee C \in I$ and $\uparrow D \wedge \mathcal{X}E \wedge F \in I$.

Let A, B, $C \in I$ and p be a path: then from Definitions 4 and 6:

$$\begin{aligned} [\![\neg \uparrow A \vee \mathcal{X}B \vee C]\!]_p &= [\![\neg\neg A \vee \neg\mathcal{X}A \vee \mathcal{X}B \vee C]\!]_p \\ &= [\![A \vee \neg\mathcal{X}A \vee \mathcal{X}B \vee C]\!]_p \\ &= [\![A \vee \neg\mathcal{X}A]\!] \vee [\![\mathcal{X}B \vee C]\!]_p. \end{aligned}$$

Therefore $\neg \uparrow A \vee \mathcal{X}B \vee C \in I$ iff $(A \vee \neg\mathcal{X}A) \vee (\mathcal{X}B \vee C) \in I$. The formula $A \vee \neg\mathcal{X}A$ is stutter invariant when only the first state in a path is stuttered. This is easy to see because it says either A is true in the first state or false in the next; if both the first state and the next state are the same, one of these must be true. $\mathcal{X}B \vee C$ is stutter invariant when any state besides the first is stuttered. This is also easy to see because the formula is only evaluated on the first state of a path. A stuttered state after the first does not change its interpretation. The interpretation of the entire formula is true as long as one part is true and together the two parts cover any instance of stuttering that could occur on a path. We have shown the formula $\neg\uparrow A \vee \mathcal{X}B \vee C$ to be stutter invariant.

Let D, E, $F \in I$. Then, $\neg E \in I$ and $\neg F \in I$. By the above, we can conclude:

$$\neg\uparrow D \vee \mathcal{X}\neg E \vee \neg F \in I$$

by (4.2),

$$\neg\big(\neg\uparrow D \vee \mathcal{X}\neg E \vee \neg F) \in I$$

by simplifying,

$$\begin{gathered} \neg\neg\uparrow D \wedge \neg\mathcal{X}\neg E \wedge \neg\neg F \in I \\ \uparrow D \wedge \mathcal{X}\neg\neg E \wedge F \in I \\ \uparrow D \wedge \mathcal{X}E \wedge F \in I. \end{gathered}$$

Finally, combining these two formulas in the form $A\mathcal{U}B$ yields our result. □

The complete proof of this theorem was given in [9]. Similarly, any up edge in the proof could be replaced with a down edge and stutter invariance would be preserved, as no step in the proof requires that the edge being used is an up edge. For simplicity up edges will be used exclusively from now on, though any of these could be replaced with a down edge for equivalently stutter invariant formulas.

Note also the following:

Property 1. If $A \in I$ and $B \in I$ and $C \in I$ then $\Diamond(\uparrow A \wedge \mathcal{X}B \wedge C) \in I$.

Proof of property 1 is analogous to that of Theorem 1. Simplified versions such as $\Diamond(\uparrow A \wedge \mathcal{X}B)$ express a simple existence property. The event represented by the up edge $\uparrow A$ must occur and then B holds.

Property 2. If $A \in I$ and $B \in I$ and $C \in I$ then $\Box(\uparrow A \rightarrow \mathcal{X}B \vee C) \in I$.

This is logically equivalent to Property 1. The simplified versions of this property, such as $\Box(\uparrow A \rightarrow \mathcal{X}B)$ express a simple universality property, that whenever the event represented by the edge $\uparrow A$ occurs, B will hold.

4.4. **LTL Patterns.** A classification of commonly used LTL formulas was made by Dwyer and colleagues. In a survey of LTL formulas used in practice, most fit into a small group of patterns, which were named and described in [3]. The goal of the pattern system is to allow reuse of formulas and show methods of constructing LTL specifications based on the characteristics being tested. Patterns can be especially useful for assisting casual users of model checking who may not be as familiar with temporal logics as many of model checking's original users. Due to the simplicity of their use and their ability to express a wide range of properties, the patterns form the basis for the formulas recognized by the LTL recognizer discussed in the next section.

Chechik and Păun [10] have extended the pattern system started by Dwyer to include edges, resulting in useful patterns of LTL formulas which are stutter invariant despite the fact that they contain the next operator. A listing of these patterns is given as an appendix.

The original pattern classifications included a category and a scope for each formula to help describe what type of property each describes. The categories considered by Chechik and Păun are:

- **Absence** - A condition, P, does not occur within a scope
- **Existence** - A condition, P, must occur within a scope
- **Universality** - A condition, P, occurs throughout a scope
- **Response** - A condition, P, must always be followed by another, S, within a scope
- **Precedence** - A condition, P, must always be preceded by another, S, within a scope

Each of the categories is associated with a scope, the region of an execution sequence on which the formula is evaluated. The five main kind of scopes used by Dwyer [3] are:

- **Global** - The entire state space
- **Before R** - The state sequence up to condition R
- **After Q** - The state sequence after condition Q
- **Between Q and R** - The part of the state sequence after condition Q and before condition R
- **After Q Until R** - Similar to between, though Q may continue to be true even if R condition does not become true.

States Q and R are used to mark the start and end of the scope for a formula. Combining the required pattern formula from the categories with the required scope formula results in an LTL formula expressing the desired property over a section of the state space.

As an example, consider the property: "The elevator door must open after stopping on a floor". It could be expressed using the Existence category with a condition, P, and the After Q scope. Let P be "the elevator door is open" and Q be "the elevator is not moving". This property can be formalized in the LTL formula $\Diamond Q \rightarrow \Diamond(Q \wedge \Diamond P)$, a stutter invariant formula.

These original patterns were state based only, meaning the conditions P and S in the categories and Q and R in the scopes were only represented as states where a

variable is true. Chechik and Păun introduced edges to the patterns in the following ways: the conditions P and S in the categories and conditions Q and R in the scope boundaries each can be either state or edge based. This allows for inclusion of events; for instance in the example above, $\uparrow P$ could express the event "The elevator door changes from a closed state to an open state" and $\downarrow Q$ could express the event "The elevator changes from a moving state to a non-moving state." Using edges in this way can better express the property in the example than a specification based only on states.

4.5. **Example Formulas Generated From Patterns.** Some more example pattern formulas with English interpretations are given below. For each example, the name of the category and scope are given, along with the meaning of each variable or edge.

Example 4.1. "The laser will not turn on while the CD tray is ejected."

P – laser is on
$\uparrow P$ – laser turns on
Q – CD tray is open
$\uparrow Q$ – CD tray opens
$R = Q$
$\downarrow R$ – CD tray closes

Pattern: Absence category with Between Q and R scope.

$$\Box((\uparrow Q \wedge \Diamond \downarrow R) \rightarrow (\neg \uparrow P \; \mathcal{U} \downarrow R))$$

4.6. **Proof Example of Pattern Stutter Invariance.** As pointed out by Chechik and Păun, using edges in patterns is only a useful concept if the resulting formulas can be shown to be stutter invariant, as they use the next operator in them.

An example proof of stutter invariance of a pattern formula is given below. This formula is a combination of the Existence category and the "between Q and R" scope.

Theorem 3.

$$\textit{If } P, Q, R \in I \textit{ then } \Box(\uparrow Q \wedge \Diamond \uparrow R \rightarrow \mathcal{X}(\neg \uparrow R \mathcal{U} P) \wedge \neg \uparrow R) \in I$$

Proof. First, to show that $\Diamond \uparrow R$ is stutter invariant, the only states which we are concerned with are the two states in which the edge $\neg R \wedge \mathcal{X} R$ occurs. By case analysis we can show that:

$$\Diamond \uparrow R \in I$$

by (4.5) and Property 1,

$$\neg \Diamond (\uparrow Q \wedge \Diamond \uparrow R \wedge \uparrow R) \in I$$

by (2.2),

$$\Box(\uparrow Q \wedge \Diamond \uparrow R \rightarrow \neg \uparrow R) \in I$$

by Theorem 1,

$$\neg \uparrow R \mathcal{U} P \in I \text{ and } \Box(\uparrow Q \wedge \Diamond \uparrow R \rightarrow \neg \uparrow R) \in I$$

by Property 1,

$$\Diamond \uparrow R \in I \text{ and } \neg \uparrow R \mathcal{U} P \in I \text{ and } \Box(\uparrow Q \wedge \Diamond \uparrow R \rightarrow \neg \uparrow R) \in I$$

by Property 2,

$$\Box(\uparrow Q \wedge \Diamond \uparrow R \rightarrow \mathcal{X}(\neg \uparrow R \mathcal{U} P)) \in I \text{ and } \Box(\uparrow Q \wedge \Diamond \uparrow R \rightarrow \neg \uparrow R) \in I$$

by (4.3),

$$\Box(\uparrow Q \wedge \Diamond \uparrow R \rightarrow \mathcal{X}(\neg \uparrow R \mathcal{U} P)) \wedge \Box(\uparrow Q \wedge \Diamond \uparrow R \rightarrow \neg \uparrow R) \in I$$

finally,

$$\Box(\uparrow Q \wedge \Diamond \uparrow R \rightarrow \mathcal{X}(\neg \uparrow R \mathcal{U} P) \wedge \neg \uparrow R) \in I$$

can be concluded by the definition of satisfaction for $\rightarrow$ and $\Box$. $\square$

These new patterns give us a system of creating and identifying stutter invariant formulas that use the next operator. No automated way to recognize these formulas was thus far available. We present an implementation of an automated way of recognizing these formulas. Full details may be found in [2].

5. **LTL Recognizer Description.** As seen in the previous sections, determining if an LTL formula is stutter invariant either requires time consuming analysis that would not be feasible for those not very familiar with temporal logic, or requires comparision with many different patterns to find a matching one. To make it easier and faster for users looking to verify properties using a model checker, a recognizer program was developed which recognizes the patterns of stutter invariant formulas we have discussed. The recognizer recognizes all the patterns identified by Chechik and Păun. Tables of these patterns are given as an appendix.

To create a program to detect certain stutter invariant formulas, Prolog was used to create the LTL recognizer. The program uses Definite Clause Grammars [8] to identify patterns that are known to be stutter invariant, in particular, those not containing the next operator, and those based on patterns and edges which have been discussed in the previous sections. The version of Prolog used was SWI-Prolog 5.2.13.

By providing Prolog with the details of what constitutes a subset of stutter invariant LTL formulas, then posing queries containing the formulas we are interested about, Prolog can report whether it follows from the rules in the program that an LTL formula is stutter invariant. Prolog's ability to determine if a query satisfies a set of rules eliminates a lot of programming that would be required to implement a similar program in an imperative programming language.

It is easy to view the formulas we are working with as strings from the LTL language. Thus it makes sense to represent the strings of the language we are interested in as a grammar. Another reason for selecting Prolog for this task is that it has a convenient method for representing grammars in its code.

Definition 7. A difference list is a list in Prolog that represents the difference between two other lists.

Difference lists can be used to represent grammars by saying that the empty list is the difference between a list coded in a program (representing the grammar rules) and the list of symbols given in a query. Prolog consumes matching symbols from the head of the lists to calculate the difference. If the difference between the user's list and the coded lists is the empty list, then all symbols matched and the user's list in the query matches some rule from the grammar difference lists.

Definition 8. A Definite Clause Grammar (DCG) is an alternate method of representing a grammar in Prolog knowledge bases. The Prolog interpreter internally translates a DCG to the corresponding difference lists. The DCG method provides a more convenient and logical method of representing grammars.

5.1. **LTL Pattern Grammar Design.** The definite clause grammar used by the LTL recognizer consists of smaller grammars, connected as a single large grammar called `invariant`. This is our Prolog predicate used for posing queries about formulas to check if the formula matches a pattern.

The main grammar connects one smaller grammar for each of the categories of formulas recognized and one to match formulas in the LTL_x subset of LTL formulas. As discussed earlier, LTL formulas not containing the next operator (LTL_x) are stutter invariant regardless of form. Thus, the LTL_x grammar simply checks whether a formula contains the next operator and reports that the formula is stutter invariant if it does not. Small grammars for each scope type are connected under the grammars for each category type. Together these grammars form the subset of formulas recognized by the program. The program visits each category grammar checking to see if the formula falls within a scope.

When the recognizer makes a pattern match in one of the grammars, a short message is output on the screen to indicate that the formula is stutter invariant, and which category/scope pattern was matched, or that the formula is in the LTL_x subset. If the recognizer does not find a match, the output is the standard Prolog message, "No". This means that the formula does not match a stutter invariant formula the recognizer knows. The formula still could be stutter invariant. A GUI front end to the recognizer could easily translate this to "Unable to determine if formula is stutter invariant" or a similar message.

5.2. **Programming Issues.** Despite the convenience of using Prolog DCGs for pattern matching, it presented some problems when writing the code to parse the formulas. Both Prolog and LTL use basic logic operators, some of which are typically represented with the exact same characters. Also, some LTL operators conflicted with Prolog's syntax. Examples of these operators are implication (`->`), and (`&&`), or (`||`) and LTL's always (`[]`) and eventually (`<>`) operators.

To avoid problems with Prolog trying to interpret rules with LTL operators as its own code, some LTL operators were replaced with other characters which have no special meaning to Prolog. A front end program to the Prolog pattern matching program easily translates the LTL formulas to and from the alternate character strings when queries are posed. Thus, the end user does not need to see the implementation details for resolving this problem.

Another challenge was ensuring that variables, particularly those used in edges, matched when they occurred more than once in the same formula. Prolog's natural variable instantiation allowed single variables in a pattern to be matched easily. Edges, however, use grammar rules separate from the main pattern rules and Prolog does not provide a simple method of maintaining the variable names from one occurrance of the rule to another inside the same pattern. This is because Prolog variable values do not persist from one call to a grammar to another. For example, the pattern formula $\Diamond\uparrow Q \rightarrow (\neg\uparrow P\ \mathcal{U}\ \uparrow Q)$ must match only if two edges of Q are used in the correct locations. Two edges of different variables should not match the pattern.

Prolog normally does not have persistent variables, so to remember the value, the `flag/3` predicate was used to store data in Prolog's *database*, a collection of persistent key-based values. The first time an edge grammar rule is used, the name of the variable used in the edge is stored. The next time the same rule is used, the variable name is recalled from the database and a match can be verified. A

`reset/0` predicate included in the recognizer resets the persistent values for the next matching attempt.

5.3. **Example LTL Recognizer Queries.** Below are some example LTL formula queries and their resulting output from the LTL recognizer. The following is given for each example: the LTL formula in normal notation, the formula expressed in a Prolog-formatted query with the replacements mentioned in the previous section, and the recognizer output after running the query. It is clear when viewing the Prolog queries that a GUI front end would be the preferable method of entering queries, due to the complex method Prolog requires for input. For this reason, a front end was written; the source code is available.

Example 5.1. Pattern: Existence category with Between Q and R scope.

$$\Box((\uparrow Q \wedge \Diamond \uparrow R) \rightarrow (\mathcal{X}(\neg \uparrow R \,\mathcal{U}\, P) \wedge \neg \uparrow R))$$

Prolog query:

```
invariant([^,"(","(",!,q,&&,o,q,&&,"<>",!,r,&&,o,r,")",$,"(",
  o,"(",!,!,r,&&,o,r,u,p,")",&&,!,!,r,&&, o,r,")",")"],[]).
```

Program output:

```
Matches 'between Q and R' existence pattern.
Yes
```

Example 5.2. This formula contains the next operator but does not match a pattern.

$$\Diamond R \rightarrow (T \wedge \mathcal{X} P)$$

Prolog query:

```
invariant(["<>",r,$,"(",t,&&,o,p,")"],[]).
```

Program output:

```
No
```

6. **Conclusion and Future Work.** The Prolog-based LTL formula recognizer presented in this paper provides a means of identifying a subset of the set of stutter invariant formulas in an automated way, where no automated way existed previously. The recognizer will output yes for all of the patterns identified by Chechik and Păun to be stutter invariant. Due to the complexity of computing whether any given formula is stutter invariant, a recognizer is a feasible method of providing run-time checking of formulas using the past efforts of others to identify formulas in this subset.

Rewriting can be employed to expand the subset of formulas identified as stutter invariant. By adding Prolog code to make small, relatively easily computed rewrites of equivalences in a query formula, a stutter invariant equivalence to a formula may be found. Future work on the recognizer is planned to incorporate rewriting.

The source code of the recognizer and front end can be downloaded from http://cs.stfx.ca/∼jeff/recognizer/.

Acknowledgements: The authors would like to thank Dr. Marsha Chechik for suggesting this problem. The authors would also like to thank the referees, whose comments and suggestions improved the presentation.

REFERENCES

[1] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*, Chapter 10. MIT Press, Cambridge, Massachusetts, 1999.

[2] Jeffrey Dallien. Automated checking for stutter invariance of LTL formulas. Undergraduate Honours Thesis, Department of Mathematics, Statistics and Computer Science, St. Francis Xavier University, Antigonish, Nova Scotia, 2004.

[3] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. ACM Press, 1998.

[4] Rob Gerth. Concise promela reference. Website, August 1997. Available: `http://spinroot.com/spin/Man/Quick.html`.

[5] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[6] G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[7] G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[8] Johan Bos, Patrick Blackburn and Kristina Striegnitz. Learn prolog now! Website, February 2003. Available: `http://www.coli.uni-sb.de/~kris/prolog-course/hmtl/`.

[9] Dimitrie O. Păun. Closure under stuttering in temporal formulas. Master's thesis, University of Toronto, Toronto, Ontario, 1999.

[10] Dimitrie O. Păun and Marsha Chechik. Events in linear-time properties. In *Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, pages 123–132. IEEE Computer Society, 1999.

[11] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Inf. Process. Lett.*, 63(5):243–246, 1997.

[12] Doron Peled, Thomas Wilke, and Pierre Wolper. An algorithmic approach for checking closure properties of ω-regular languages. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119, pages 596–610, Pisa, Italy, 1996. Springer.

Appendix A. **Tables of Recognized LTL Patterns.** Under each scope, formulas are numbered depending on which combination of states and edges are used. The combinations are:

0 P, S - states, Q, R - states
1 P, S - states, Q, R - edges
2 P, S - edges, Q, R - states
3 P, S - edges, Q, R - edges

To conserve space in this listing, the following equivalences are used:

$$A \mathcal{W} B \equiv \Box A \vee (A \mathcal{U} B) \quad \text{(weak until)}$$
$$A \mathcal{P} B \equiv \neg(\neg A \mathcal{U} B) \quad \text{(precedes)}$$
$$\text{if } A \text{ then } B \text{ else } C \equiv (A \rightarrow B) \wedge (\neg A \rightarrow C)$$

The recognizer uses the expanded forms as Spin does not use these equivalences. These tables were originally given in [9].

Absence Pattern

Globally	Before R
0 $\Box\neg P$	0 $\Diamond R \rightarrow (\neg P \mathcal{U} R)$
1 $\Box\neg P$	1 $\Diamond \uparrow R \rightarrow (\uparrow R \mathcal{P} P)$
2 $\Box\neg \uparrow P$	2 $\Diamond R \rightarrow (\neg \uparrow P \mathcal{U} R)$
3 $\Box\neg \uparrow P$	3 $\Diamond \uparrow R \rightarrow (\neg \uparrow P \mathcal{U} \uparrow R)$
After Q	Between Q and R
0 $\Box(Q \rightarrow \Box\neg P)$	0 $\Box((Q \wedge \Diamond R) \rightarrow (\neg P \mathcal{U} R))$
1 $\Box(\uparrow Q \rightarrow \mathcal{X}\Box\neg P)$	1 $\Box((\uparrow Q \wedge \Diamond \uparrow R \wedge \neg \uparrow R) \rightarrow \mathcal{X}(\uparrow R \mathcal{P} P))$
2 $\Box(Q \rightarrow \Box\neg \uparrow P)$	2 $\Box((Q \wedge \Diamond R) \rightarrow (\neg \uparrow P \mathcal{U} R))$
3 $\Box(\uparrow Q \rightarrow \Box\neg \uparrow P)$	3 $\Box((\uparrow Q \wedge \Diamond \uparrow R) \rightarrow (\neg \uparrow P \mathcal{U} \uparrow R))$
After Q until R	
0 $\Box((Q \wedge \Diamond P) \rightarrow (\neg P \mathcal{U} R))$	
1 $\Box((\uparrow Q \wedge \neg \uparrow R \wedge \mathcal{X}\Diamond P) \rightarrow \mathcal{X}(\uparrow R \mathcal{P} P))$	
2 $\Box(Q \rightarrow (\neg P \mathcal{W} R))$	
3 $\Box(\uparrow Q \rightarrow (\neg \uparrow P \mathcal{W} \uparrow R))$	

Existence Pattern

Globally	Before R
$0\ \Diamond P$	$0\ \Diamond R \rightarrow (P\ \mathcal{P}\ R)$
$1\ \Diamond P$	$1\ \Diamond \uparrow R \rightarrow (\neg \uparrow R\ \mathcal{U}\ P)$
$2\ \Diamond \uparrow P$	$2\ \Diamond R \rightarrow (\uparrow P\ \mathcal{P}\ R)$
$3\ \Diamond \uparrow P$	$3\ \Diamond \uparrow R \rightarrow (\uparrow P\ \mathcal{P}\ \uparrow R)$
After Q	**Between Q and R**
$0\ \Diamond Q \rightarrow \Diamond(Q \wedge \Diamond P)$	$0\ \Box((Q \wedge \Diamond R) \rightarrow ((P\ \mathcal{P}\ R) \wedge \neg R))$
$1\ \Diamond \uparrow Q \rightarrow \Diamond(\uparrow Q \wedge \mathcal{X}\Diamond P)$	$1\ \Box((\uparrow Q \wedge \Diamond \uparrow R) \rightarrow (\mathcal{X}(\neg \uparrow R\ \mathcal{U}\ P) \wedge \neg \uparrow R))$
$2\ \Diamond Q \rightarrow \Diamond(Q \wedge \Diamond \uparrow P)$	$2\ \Box((Q \wedge \Diamond R) \rightarrow ((\uparrow P\ \mathcal{P}\ R) \wedge \neg R))$
$3\ \Diamond \uparrow Q \rightarrow \Diamond(\uparrow Q \wedge \Diamond \uparrow P)$	$3\ \Box((\uparrow Q \wedge \Diamond \uparrow R) \rightarrow ((\uparrow P\ \mathcal{P}\ \uparrow R) \wedge \neg \uparrow R))$
After Q Until R	
$0\ \Box(Q \rightarrow (\text{if } \Diamond R \text{ then } ((P\ \mathcal{P}\ R) \wedge \neg R) \text{ else } \Diamond P))$	
$1\ \Box(\uparrow Q \rightarrow \mathcal{X}(\neg \uparrow R\ \mathcal{U}\ P) \wedge \neg \uparrow R)$	
$2\ \Box(Q \rightarrow (\text{if } \Diamond R \text{ then } ((\uparrow P\ \mathcal{P}\ R) \wedge \neg R) \text{ else } \Diamond \uparrow P))$	
$3\ \Box(\uparrow Q \rightarrow (\text{if } \Diamond \uparrow R \text{ then}((\uparrow P\ \mathcal{P}\ \uparrow R) \wedge \neg \uparrow R) \text{ else } \Diamond \uparrow P))$	

Precedence Pattern

Globally	Before R
$0\ \Diamond P \rightarrow S\ \mathcal{P}\ P$	$0\ \Diamond R \rightarrow (\neg P\ \mathcal{U}\ ((S \wedge \neg P) \vee R))$
$1\ \Diamond P \rightarrow S\ \mathcal{P}\ P$	$1\ \Diamond\ \uparrow R \rightarrow ((\neg \uparrow R\ \mathcal{U}\ P) \rightarrow (S\ \mathcal{P}\ P))$
$2\ \Diamond \uparrow P \rightarrow \uparrow S\ \mathcal{P}\ \uparrow P$	$2\ \Diamond R \rightarrow (\neg \uparrow P\ \mathcal{U}\ ((\uparrow S \wedge \neg \uparrow P) \vee R))$
$3\ \Diamond \uparrow P \rightarrow \uparrow S\ \mathcal{P}\ \uparrow P$	$3\ \Diamond \uparrow R \rightarrow ((\uparrow P\ \mathcal{P}\ \uparrow R) \rightarrow (\uparrow S\ \mathcal{P}\ \uparrow P))$
After Q	
$0\ \Diamond Q \rightarrow \Diamond(Q \wedge (\Diamond P \rightarrow (S\ \mathcal{P}\ P)))$	
$1\ \Diamond \uparrow Q \rightarrow \Diamond(\uparrow Q \wedge \mathcal{X}(\Diamond P \rightarrow (S\ \mathcal{P}\ P)))$	
$2\ \Diamond Q \rightarrow \Diamond(Q \wedge (\Diamond \uparrow P \rightarrow (\uparrow S\ \mathcal{P}\ \uparrow P)))$	
$3\ \Diamond \uparrow Q \rightarrow \Diamond(\uparrow Q \wedge (\Diamond \uparrow P \rightarrow (\uparrow S\ \mathcal{P}\ \uparrow P)))$	
Between Q and R	
$0\ \Box((Q \wedge \Diamond R) \rightarrow (\neg P\ \mathcal{U}\ ((S \wedge \neg P) \vee R)))$	
$1\ \Box((\uparrow Q \wedge \neg \uparrow R\ \mathcal{X}\Diamond \uparrow R) \rightarrow \mathcal{X}((\neg \uparrow R\ \mathcal{U}\ P) \rightarrow (S\ \mathcal{P}\ P)))$	
$2\ \Box((Q \wedge \Diamond R) \rightarrow (\neg \uparrow P\ \mathcal{U}\ ((\uparrow S \wedge \neg \uparrow P) \vee R)))$	
$3\ \Box((\uparrow Q \wedge \neg \uparrow R\ \mathcal{X}\Diamond \uparrow R) \rightarrow \mathcal{X}((\uparrow P\ \mathcal{P}\ \uparrow R) \rightarrow (\uparrow S\ \mathcal{P}\ \uparrow P)))$	
After Q until R	
$0\ \Box(Q \rightarrow (\Diamond P \rightarrow \neg P\ \mathcal{U}\ ((S \wedge \neg P) \vee R)))$	
$1\ \Box(\uparrow Q \rightarrow \mathcal{X}(\Diamond P \rightarrow ((\neg \uparrow R\ \mathcal{U}\ P) \rightarrow (S\ \mathcal{P}\ P))))$	
$2\ \Box(Q \rightarrow (\Diamond P \rightarrow \neg \uparrow P\ \mathcal{U}\ ((\uparrow S \wedge \neg \uparrow P) \vee R)))$	
$3\ \Box(\uparrow Q \rightarrow \mathcal{X}(\Diamond P \rightarrow ((\uparrow P\ \mathcal{P}\ \uparrow R) \rightarrow (\uparrow S\ \mathcal{P}\ \uparrow P))))$	

Response Pattern

Globally	Before R
$0\ \Box(P \to \Diamond S)$	$0\ \Diamond R \to (P \to (\neg R\ \mathcal{U}\ S))\ \mathcal{U}\ R$
$1\ \Box(P \to \Diamond S)$	$1\ \Diamond \uparrow R \to ((P \to (\neg \uparrow R\ \mathcal{U}\ S)) \wedge \neg \uparrow R)\ \mathcal{U}\ (\uparrow R \wedge (P \to Q))$
$2\ \Box(\uparrow P \to \Diamond \uparrow S)$	$2\ \Diamond R \to (\uparrow P \to (\neg R\ \mathcal{U}\ \uparrow S))\ \mathcal{U}\ R$
$3\ \Box(\uparrow P \to \Diamond \uparrow S)$	$3\ \Diamond \uparrow R \to ((\uparrow P \to (\neg \uparrow R\ \mathcal{U}\ \uparrow S))\ \mathcal{U}\ \uparrow R)$
After Q	
$0\ \Box(Q \to \Box(P \to \Diamond S))$	
$1\ \Box(\uparrow Q \to \mathcal{X}\Box(P \to \Diamond S))$	
$2\ \Box(Q \to \Box(\uparrow P \to \Diamond \uparrow S))$	
$3\ \Box(\uparrow Q \to \Box(\uparrow P \to \Diamond \uparrow S))$	
Between Q and R	
$0\ \Box((Q \wedge \Diamond R) \to ((P \to (\neg R\ \mathcal{U}\ S))\ \mathcal{U}\ R))$	
$1\ \Box((\uparrow Q \wedge \Diamond \uparrow R \wedge \neg \uparrow R) \to \mathcal{X}(((P \to (\neg \uparrow R\ \mathcal{U}\ S)) \wedge \neg \uparrow R)\ \mathcal{U}\ (\uparrow R \wedge (P \to Q))))$	
$2\ \Box((Q \wedge \Diamond R) \to ((\uparrow P \to (\neg R\ \mathcal{U}\ \uparrow S))\ \mathcal{U}\ R))$	
$3\Box((\uparrow Q \wedge \Diamond \uparrow R) \to ((\uparrow P \to (\neg \uparrow R\ \mathcal{U}\ \uparrow S))\ \mathcal{U}\ \uparrow R))$	
After Q until R	
$0\ \Box(Q \to (P \to (\neg R\ \mathcal{U}\ S))\ \mathcal{W}\ R)$	
$1\ \Box(\uparrow Q \to \mathcal{X}(((P \to (\neg \uparrow R\ \mathcal{U}\ S)) \wedge \neg \uparrow R)\ \mathcal{W}\ (\uparrow R \wedge (P \to Q))))$	
$2\ \Box(\uparrow Q \to (\uparrow P \to (\neg R\ \mathcal{U}\ \uparrow S))\ \mathcal{W}\ R)$	
$3\ \Box(\uparrow Q \to (\uparrow P \to (\neg \uparrow R\ \mathcal{U}\ \uparrow S))\ \mathcal{W}\ \uparrow R)$	

Universal Pattern
(Note: P cannot be an edge because the resulting formula is always false.)

Globally	Before R
$0\ \Box P$	$0\ \Diamond R \to (P\ \mathcal{U}\ R)$
$1\ \Box P$	$1\ \Diamond \uparrow R \to (\uparrow R\ \mathcal{P}\ \neg P)$
After Q	Between Q and R
$0\ \Box(Q \to \Box P)$	$0\ \Box((Q \wedge \Diamond R) \to (P\ \mathcal{U}\ R))$
$1\ \Box(\uparrow Q \to \mathcal{X}\Box P)$	$1\ \Box((Q \wedge \Diamond \uparrow R \wedge \neg\uparrow R) \to \mathcal{X}(\uparrow R\ \mathcal{P}\ \neg P))$
After Q Until R	
$0\ \Box(Q \to P\ \mathcal{W}\ R)$	
$1\ \Box(\uparrow Q \to$ (if $\Diamond \uparrow R$ then $(\mathcal{X}(\uparrow R\ \mathcal{P}\ \neg P))$ else $\Box P))$	

Received November 2005.

E-mail address: jdallien@stfx.ca; wmaccaul@stfx.ca