PROGRAMMING REACTIVE SYSTEMS IN TEMPORAL LOGIC

by

Alexander Tuzhilin
Information Systems Department
Leonard N. Stern School of Business
New York University
40 West 4th Street, Room 624
New York, New York 10003

October, 1990

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-90-21

Programming Reactive Systems in Temporal Logic

Alexander Tuzhilin

Information Systems Department
Stern School of Business
New York University
40 West 4th Street, Room 624
New York, NY 10003
e-mail: tuzhilin@rnd.gba.nyu.edu

1 Introduction

There are many important systems that evolve indefinitely long in time. Examples of these systems comprise operating, process control, communication, and flexible manufacturing systems (FMSes) [Gro87, Ran86]. Pnueli calls such systems *reactive systems* [Pnu86] and proposes temporal logic as a specification language for reactive systems.

However, temporal logic is used not only as a specification language. There have been several proposals to use predicate temporal logic as a programming language [AM89, Bau89, BFG+89, KKN+90, FKTMo86, Mos86]. Each of these proposals selects a certain type of temporal logic and defines a semantics of programs in the resulting language.

METATEM [BFG+89] and TEMPLOG [AM89, Bau89] provide two diametrically opposite approaches to the design of a language based on temporal logic. METATEM is a very expressive language. In fact, Gabbay shows in the Separation Theorem [Gab89] that the propositional counterpart of METATEM has the power of full propositional temporal logic. However, the semantics of METATEM is *imperative* [BFG+89], i.e. very procedural. Furthermore, some METATEM programs have non-deterministic computations resulting in backtracking. This leads to computations whose complexity grows exponentially in time.

Contrary to METATEM, TEMPLOG has limited expressive power since it does not support negation. On the other hand, it has a well-defined declarative semantics [Bau89] and a computational procedure based on temporal resolution [AM89].

In this paper we take a middleground approach to the design of a language based on temporal logic. We want to find a fragment of temporal logic such that

- behavior of reactive systems *can* be modeled in this fragment and can be modeled in a *"natural"* way;
- the language has a well-defined declarative semantics;
- programs written in this language have efficient computations, i.e. are linear in time (e.g. no backtracking).

In other words, we want to find a fragment of temporal logic with the right balance between expressive power on one hand and a clear meaning and computational efficiency on the other hand which is *suitable* for programming reactive systems.

We contribute to the solution of this problem by presenting the language PTL[1] based on a fragment of predicate temporal logic with inflationary semantics [GS86, AV88, KP88]. The language will be formally defined in Section 3. However, before that, we state the requirements that a programming language for reactive systems must satisfy.

2 Requirements for Programming Reactive Systems

In this paper we propose a certain fragment of temporal logic as a programming language for reactive systems. In order to motivate the selection of this specific fragment, we consider a Flexible Manufacturing System [Ran86, Gro87] as an important example of a reactive system and see what requirements it imposes on a fragment of temporal logic to be used for programming reactive systems.

Example 1 A Flexible Manufacturing System (FMS) manufactures certain products such as car engines, electronic boards, or electrical appliances. In this example, we assume that an FMS performs assembly operations on unfinished units. The initial part of an assembly is brought into the system through the *load-unload* station. Then it is carried among various manufacturing units, called *cells*, where the assembly process takes place. For example, in case an FMS manufactures toasters, one cell can be responsible for making the outer body of a toaster, another for installing heating elements in it, another for assemblying knobs on

[1] PTL stands for Programming in Temporal Logic.

the front panel of a toaster, and still another one for attaching the front door to it. A special vehicle, called an *Automatic Guidance Vehicle (AGV)*, carries incomplete assemblies among various cells. When the assembly process is completed, the finished units are brought by AGVs back to the load-unload station where they are removed from the FMS system.

■

An FMS, as described in Example 1, has certain features common to other reactive systems. They must be properly supported in temporal logic for it to be a good programming language for reactive systems. We describe the features below and explain what requirements they impose on a fragment of temporal logic for it to be well suited for programming reactive systems.

First, an FMS system evolves forward in time from the past into the future. Therefore, a fragment of temporal logic for programming reactive systems must support this forward movement in time. This means that the formulas of the type $\circ p \rightarrow q$ describing backward movement in time are unacceptable in the language (note that these formulas are allowed in TEMPLOG [AM89]).

Second, an FMS is a very dynamic system. Conditions that hold now may not hold in the future. For example, if a vehicle is located at a cell now, it may not be located there in the future because it may move elsewhere. This means that the language must explicitly support updates. By that we mean that if a certain statement is true now, it may not be true in the future as the previous example shows.

Third, certain conditions may hold only for some time in the future or can happen within a limited time frame. For example, a vehicle can travel between two cells only for a certain amount of time. This means that a temporal logic should support statements like "the condition will always be true in the future *within* some time frame."

Fourth, we may want to ask queries about an FMS system. For example, we may want to know which cells a vehicle will visit within the next 60 minutes. This means that the language must support program calls, and these calls should also be expressed in temporal logic.

Fifth, an FMS system exhibits non-deterministic behavior. For example, a vehicle can travel from a given cell to *several* other cells where the next assembly operation can take place. This requires a non-deterministic choice of a cell to which a vehicle must travel. Therefore, temporal logic should also support non-determinism. This requirement is consistent with

the definition of a reactive system [Pnu86], where a transition function maps a state into a *set* of states, and a specific computation makes a non-deterministic choice among the various "next" states.

Note that these requirements are applicable not only to FMSes but also to other reactive systems such as operating systems and communication networks. In the next section, we propose a fragment of a predicate temporal logic that satisfies these requirements.

3 PTL: Programming in Temporal Logic

In this section, we define the syntax and the semantics of PTL language that satisfies the requirements for programming reactive systems as stated in Section 2.

3.1 Syntax of PTL

Before defining the syntax of PTL, we explain how the language will satisfy the requirements introduced in Section 2.

The first requirement of Section 2 will be supported by restricting temporal logic formulas to the clausal form $BODY \rightarrow HEAD$, where $BODY$ refers to the past and present and $HEAD$ to the future. This restriction provides for the forward movement in time. The second requirement will be supported by allowing negations both in the head and in the body of a rule. This feature supports updates by making some conditions false after they have been true. The third requirement will be supported by the introduction of the bounded necessity and possibility operators and their past mirror images. These temporal operators will be defined below. The fourth requirement will be supported by the introduction of a query language in Section 4. A program call will be interpreted as a query in this language. The fifth requirement will be supported by the introduction of *non-deterministic variables.*

To define PTL formally, we introduce the following preliminary concepts. First, we use standard *future* temporal operators of *necessity* $\Box$, *possibility* $\Diamond$, and *next* $\circ$ [Kro87, RU71]. $\Box A$ is true if A is always true in the future, $\Diamond A$ is true if A is true at some time in the future, and $\circ A$ is true if A is true at the next moment of time. We also use past mirror images of these operators: *past necessity* $\blacksquare$, *past possibility* $\star$, and *previous* $\odot$. Second, we introduce two new temporal operators of *bounded necessity* $\Box_T$ and *bounded possibility* $\Diamond_T$ together with their past mirror images. $\Box_T A$ is true at time t if A is true from time t up

to but not including $t+T$ and is false at time $t+T$. $\diamond_T A$ is true at time t if there is t', such that $t \leq t' \leq t+T$ and A is true at t'. We need such bounded operators to be able to express bounded temporal properties of reactive systems, e.g. the statements such as "an AGV is moving from one cell to another *within* a certain time period." Additional examples of bounded temporal operators will be provided in Example 2.

We define *literals* as in the standard first-order logic [End72]: a literal is either an atomic formula or a negated atomic formula. A *next-literal* (in analogy with *next-atom* of [AM89]) is a literal preceded by a finite number of *next* operators $\circ$. Similarly, a *previous-literal* is a literal preceded by a finite number of *previous* operators.

We are now ready to define the syntax of PTL. A *PTL program* is a set of temporal clauses. A temporal clause has the form $BODY \rightarrow HEAD$, where $BODY$ is *any* temporal logic formula with only the *past* temporal operators appearing in them; and $HEAD$ is a conjunction of next-literals and necessity and bounded necessity operators. Notice that we make another syntactic restriction assuming that the head of a rule does not have temporal possibility operators. This restriction becomes important when we consider the semantics of PTL. As Example 2 shows, conjunctions in the head of a rule will be denoted with semicolon (;). It follows from this definition that the body of a rule refers to the current moment of time and to the past, whereas the head of a rule refers strictly to the future. In addition, PTL supports negations both in the head and the body of a rule.

Variables in temporal clauses are divided into *deterministic* and *non-deterministic* [Tuz89]. The distinction between these two types of variables will be intuitively explained in Example 2 and will be formally defined in Section 3.2.

Next, we provide examples of PTL rules and explain various points about PTL using these examples.

Example 2 Behavior of an FMS system presented in Example 1 can be described with PTL rules. We present only several rules providing a partial program describing FMS behavior. The state of an FMS system is defined with the following predicates. $D(AGV, C)$: a vehicle AGV is docked at a cell C; $L1(ASM, AGV)$: an assembly ASM is loaded on a vehicle AGV; $L2(ASM, C)$: an assembly ASM is located in a cell C; $NEXT(C, C')$: the next assembly operation is done in cell C' after the previous assembly operation is done in cell C; $TRAV(C, C', T)$: it takes T units of time for an AGV to travel from cell C to cell C'; $PROC(C, T)$: it takes T units of time to perform an operation in cell C.

Non-deterministic variables are overlined in the rules. Their meaning will be explained after the rules are presented together with some other comments about this program.

R1: If an AGV is docked at a cell with an assembly on it and the assembly has been processed by the cell, then move the AGV to *one of* the next cells for the time period determined by relation $TRAV$.

$D(AGV, C) \wedge L1(ASM, AGV) \wedge \star L2(ASM, C) \wedge NEXT(C, \overline{C'}) \wedge TRAV(C, \overline{C'}, T) \rightarrow$
$\circ\neg D(AGV, C); \Box_T MOV(AGV, \overline{C'})$

R2: If an AGV arrived at a cell then dock it at that cell.

$\odot MOV(AGV, C) \wedge \neg MOV(AGV, C) \rightarrow \circ D(AGV, C)$

R3: If some AGV's are docked at a cell with assemblies loaded on them that have not been processed by the cell yet, and no other assembly is in that cell, then transfer the assembly from *one of* the AGV's to the cell.

$D(\overline{AGV}, C) \wedge L1(\overline{ASM}, \overline{AGV}) \wedge \neg \star L2(\overline{ASM}, C) \wedge (\forall ASM')\neg L2(ASM', C)$
$\wedge PROC(C, T) \rightarrow \circ\neg L1(\overline{ASM}, \overline{AGV}); \Box_T L2(\overline{ASM}, C)$

R4: If an operation on an assembly is finished by the cell and an empty AGV is docked at the cell then put the assembly on the AGV.

$\odot L2(ASM, C) \wedge \neg L2(ASM, C) \wedge D(\overline{AGV}, C) \wedge (\forall ASM')\neg L1(ASM', \overline{AGV})$
$\rightarrow \circ L1(ASM, \overline{AGV})$

■

Comments about Example 2 are in order. First, notice the usage of non-deterministic variables. For example, the rule **R1** says that if an AGV can be moved to several possible cells where the next operation can take place then *non-deterministically* select only one of them. Therefore, PTL supports future non-deterministic scenarios which correspond to the branching version of temporal logic [RU71]. This choice is facilitated by the non-deterministic variable $\overline{C'}$. More precise meaning of non-deterministic variables will be provided in Section 3.2. Second, notice the usage of bounded and unbounded possibility and necessity operators. For example, rule **R1** says that if an assembly has been located in a cell *in the past*, i.e. the assembly has been processed by the cell (unbounded past possibility operator), then keep moving an AGV to the next cell for a *specified* amount of time (bounded necessity operator). Third, the heads of rules contain only bounded and unbounded future necessity operators,

and the bodies of the rules contain bounded and unbounded past possibility and necessity operators. Therefore, the body of a rule refers to the past, and the head refers to the future. This is a restriction of the language defined above. Finally, observe that the predicates $TRAV$, $NEXT$, and $PROC$ do not appear in the head of any rule and, therefore, do not change over time. They are called *rigid* predicates [AM89].

A PTL program is *normalized* if the body of any of its rules is a conjunction of literals. Clearly, normalized programs form a subset of PTL. However, it can be shown using techniques from [Tuz89] that for any unnormalized PTL program, there exists an equivalent normalized PTL program.

3.2 Semantics of PTL

Baudinet [Bau89] defined a model-theoretic semantics of a TEMPLOG program [AM89]. However, this semantics cannot be extended to PTL because PTL supports negations, whereas Baudinet's semantics does not. Therefore, PTL requires a different type of semantics. We consider the deterministic case at first, i.e. we assume that all the variables in a program are deterministic; we extend the semantics to the non-deterministic case later on.

Semantics of a PTL program is based on the concept of a *temporal interpretation* and a *temporal model* of a temporal logic formula [RU71]. A temporal interpretation of a program defines the domain of discourse, the model of time (e.g. discrete or continuous, bounded or unbounded, linear or branching), assigns values to constants, predicates and function symbols as in classical logic, and specifies the values of all the predicates in the program at *all* time instances. We assume any arbitrary structure of the domain of discourse, assume that time is defined with integers, and assume that the domain of discourse *does not change* over time. A temporal interpretation is a *temporal model* of a PTL program if all the clauses of that program are true at all the time instances in that interpretation. If the domain of discourse is the Herbrand universe [End72], i.e. it consists of all the ground next-atomic formulas of the form $\circ^k p$, then the model is called *temporal Herbrand model.*

Example 3 Consider the program $\neg p \rightarrow \circ\, p$ expressed in the propositional temporal logic. It has infinitely many temporal Herbrand models. Some of the examples of these models are

1. $\{\circ^i p\}$ for $i = 0, 1, 2, 3, \ldots$
2. $\{\circ^i p\}$ for $i = 1, 3, 5, 7, \ldots$

3. $\{\circ p,\ \circ^2 p,\ \circ^i p\}$ for $i = 4, 5, 6, \ldots$

where $\circ^i$ is the composition of *next* $\circ$ operator applied i times.

■

As Example 3 shows, there can be many models of a temporal logic program in general. Therefore, we have to provide some way to select a canonical model out of the set of all the possible models to define semantics of a PTL program. We propose the following *inflationary conditions* to restrict the set of models of a program. Intuitively, inflationary conditions say that if a fact is not explicitly removed from the set of valid facts by application of rules at the current moment of time then it remains valid at the next moment of time.

Formally, let $P_1, P_2, \ldots, P_m$ be the set of the predicates in a PTL program that change over time, i.e. the predicates that appear in the head of some rule. Let D_i be the vector of instances of predicates $P_1, P_2, \ldots, P_m$ at time i, i.e. $D_i = (P_{1i}, P_{2i}, \ldots, P_{mi})$. We say that a model of a program satisfies *inflationary conditions* if values of predicates $P_1, P_2, \ldots, P_m$ at all the time instances i.e. $D_0, D_1, D_2, \ldots$, satisfy the following requirements. Let $IN(D_0, \ldots, D_k)$ and $OUT(D_0, \ldots, D_k)$ be the sets of the ground atoms that have to be true and false respectively at time $k+1$ in the model that has $D_0, D_1, \ldots, D_k$ as instances of its predicates at times $0, 1, \ldots, k$[2]. In other words, IN and OUT determine the ground atoms that will be, respectively, added to and removed from the predicates at the "next" time moment $k+1$, assuming the "history" of these predicates was $D_0, D_1, \ldots, D_k$. Then the model satisfies the inflationary conditions if $D_{k+1} = D_k \cup IN(D_0, \ldots, D_k) - OUT(D_0, \ldots, D_k)$. In other words, if the ground atom $P_i(a_1, \ldots, a_n)$ is true at time k, and the model does not require $P_i(a_1, \ldots, a_n)$ to be false at time $k+1$ then $P_i(a_1, \ldots, a_n)$ is true at time $k+1$.

Example 4 Consider the program from Example 3. As was shown in Example 3, this program can have infinitely many temporal Herbrand models. However, it can have only two temporal Herbrand models satisfying inflationary conditions:

1. $\{\circ^i p\}$ for $i = 0, 1, 2, \ldots$
2. $\{\circ^i p\}$ for $i = 1, 2, 3, \ldots$.

■

[2]Observe that $IN(D_0, \ldots, D_k) \cap OUT(D_0, \ldots, D_k) = \emptyset$ must hold in order that the temporal interpretation based on the temporal structure $D_0, D_1, D_2, \ldots$ be a model.

Inflationary conditions can be traced back to the inflationary operators of Gurevich and Shelah [GS86] and to the inflationary semantics of negated Datalog programs [AV88, KP88].

Semantics of PTL programs defined with inflationary conditions is important because of the following reasons. First, such semantics appears to be "natural" for reactive systems. Intuitively, temporal logic programs describe *changes* to a reactive system. Whenever changes do not occur, components of the system remain unchanged which is exactly what inflationary conditions say. For example, if a rule makes an AGV move to another cell if certain conditions are satisfied, then other AGVs that do not satisfy these conditions stay where they are. Second, as related to the first point, inflationary conditions provide a solution to the *frame problem* [MH69] for PTL programs: if the ground atom does not become false at the next time instance, it is assumed to be true at that time. Third, inflationary semantics makes PTL at least as powerful as production systems with the "in case of conflicts, cancel the computation" conflict resolution strategy. It can be shown that the inflationary semantics of PTL programs in the degenerate case when rules can have only *next* temporal operators in their heads coincides with the semantics of production systems with the aforementioned conflict resolution strategy. For example, the temporal clause $P(x) \rightarrow \circ Q(x)$ with the inflationary semantics is equivalent to the production rule $P(x) \rightarrow INSERT(Q, x)$, and the clause $P(x) \rightarrow \circ\neg Q(x)$ is equivalent to the rule $P(x) \rightarrow DELETE(Q, x)$.

In case non-deterministic variables are allowed in a program, the semantics of a program is associated with a set of non-deterministic trajectories. Each trajectory is defined by the inflationary semantics and by specific choices of non-deterministic variables. More specifically, sets $IN(D_0, \ldots, D_k)$ and $OUT(D_0, \ldots, D_k)$ of trajectory $D_0, D_1, D_2, \ldots$, are determined at time k by instantiating deterministic variables in rule bodies based on states $D_0, D_1, \ldots, D_k$ and making choices among values of non-deterministic variables corresponding to the instantiated deterministic variables. A specific choice determines the next state of a non-deterministic trajectory. As before, the inflationary semantics for a non-deterministic trajectory assumes that $D_{k+1} = D_k \cup IN(D_0, \ldots, D_k) - OUT(D_0, \ldots, D_k)$.

However, the inflationary conditions still do not define the unique model of a PTL program. As follows from Example 4, the program $\neg p \rightarrow \circ\, p$ still has two models. This motivates other types of restrictions on the possible models of a PTL program called *boundary conditions*[3].

We distinguish between the two types of boundary conditions. First, the *initial condi-*

[3]In analogy to boundary conditions for a solution of partial differential equations.

$NEXT(LU, C1)$	$TRAV(LU, C1, 5)$	$PROC(LU, 1)$
$NEXT(LU, C2)$	$TRAV(LU, C2, 8)$	$PROC(C1, 3)$
$NEXT(C1, C2)$	$TRAV(C1, C2, 3)$	$PROC(C2, 7)$
$NEXT(C1, C3)$	$TRAV(C1, C3, 6)$	$PROC(C3, 5)$
$NEXT(C2, C3)$	$TRAV(C2, C3, 4)$	
$NEXT(C3, LU)$	$TRAV(C3, LU, 6)$	
NEXT	**TRAV**	**PROC**

Figure 1: Initial Instances of Rigid Predicates

tions define predicate instances at time $t = 0$. For example, if we assume that the value of p is *TRUE* at time $t = 0$ then the program from Example 3 has the unique model $\{\circ^i p\}$ for $i = 0, 1, 2, 3, \ldots$. The second type of a boundary condition specifies values of the predicates that do not appear in the head of any rule in a program. These predicates do not change over time and are called *rigid* in [AM89]. The meaning of these two types of boundary conditions will be illustrated by the following example.

Example 5 The PTL program from Example 2 has the following boundary conditions. First, it has rigid predicates *NEXT, TRAV*, and *PROC* that do not change over time. Possible examples of their values are shown in Fig. 1. Second, initial conditions must be specified for predicates *D, L1, L2*, and *MOV*. These initial conditions are determined by the initial configuration of the FMS system and are left unspecified in this example.

The two types of boundary conditions uniquely specify the model of the PTL program from Example 2 assuming inflationary semantics and some choice of non-deterministic variables.

∎

The inflationary semantics combined with the boundary conditions suggests a computational mechanism to construct the trajectory of a given program. If only the deterministic case is considered then, in its naive implementation, the next state of the trajectory is obtained from the previous one by adding newly derived facts and removing the obsolete facts. This means that the n-th state in the trajectory can be computed in time linear in n for the deterministic case.

3.3 Relationship to Temporal Prolog

As was stated in the introduction, METATEM and TEMPLOG provide two diametrically opposite approaches to the design of a programming language based on temporal logic. In contrast to this, PTL takes a middleground approach: it selects a fragment of temporal logic that is powerful enough to be well-suited for programming reactive systems and that also has a declarative semantics and is computationally efficient.

Temporal Prolog, as part of RACCO programming language [KKN+90], also takes a middleground approach. It forms a subset of temporal logic with negations allowed in the body but not in the head of a rule. Also, it has a declarative semantics supporting efficient computations.

However, PTL differs from Temporal Prolog in the following respects. First, it supports negation not only in the body of a rule but also in its head. Second, PTL and Temporal Prolog have different semantics. Semantics of Temporal Prolog assumes that if the fact is not made true explicitly at the next time moment, it is assumed to be false, whereas the inflationary semantics says that if a fact is not made false explicitly at the next time moment, it is assumed to be true. Clearly, the inflationary semantics is better in the case when the number of changes to the set of facts in one step is smaller than the total number of facts. This is often the case in reactive systems. Third, the two languages have different types of temporal operators. However, this is not an important distinction because any Temporal Prolog and any PTL program without bounded operators can be reduced to an equivalent program with only *next* temporal operator [KKN+90]. Fourth, PTL supports non-determinism. Fifth, as described in Section 4, we define calls to PTL programs in terms of arbitrary temporal logic formulas, whereas Temporal Prolog does not.

4 Temporal Logic Queries

In this section, we define *calls* to PTL programs. The calls will be associated with *queries* against temporal logic programs defined in Section 3. In accordance with requirement 4, as stated in Section 2, the queries should also be expressed in a predicate temporal logic.

We make a restrictive assumption that temporal logic programs do not contain non-deterministic variables. In other words, we consider only the deterministic case. Non-deterministic extensions to such a query language for the programs defined with production

systems were studied in [Tuz89].

A *temporal logic query* on a PTL program P is an expression of the form

$$\{x_1, x_2, \ldots, x_n \mid \phi(x_1, x_2, \ldots, x_n)\}$$

where ϕ is a predicate temporal logic formula with all of its predicates appearing in program P, and $x_1, x_2, \ldots, x_n$ are some of the free variables in ϕ.

The answer to this query is the set of tuples $(x_1, x_2, \ldots, x_n)$ for which the formula $\phi(x_1, x_2, \ldots, x_n)$ is true *at present* time moment for the temporal model determined by program P.

Example 6

Q1: Find all the assemblies that will pass through the cell C_0 within the next 60 min.

$$\{ASM \mid \diamond_{60}\, L2(ASM, C_0)\}$$

$\diamond_{60}$ is the bounded possibility operator introduced in Section 3.1.

Q2: Find all the assemblies that will stay in the cell C_0 exactly throughout the next 60 minutes.

$$\{ASM \mid \Box_{60} L2(ASM, C_0)\}$$

where $\Box_{60}$ is the bounded necessity operator.

Q3: Find all the assemblies that will be produced within the next hour.

$$\{ASM \mid \diamond_{60}\, (L2(ASM, LU) \wedge \star L2(ASM, C) \wedge C \neq LU)\}$$

The predicates inside the parenthesis in the query say that the ASM is at the load/unload station now but it is the finished assembly (as opposed to the newly introduced one) because it was in the FMS system in the past.

∎

An arbitrary predicate temporal logic query is undecidable in general. Therefore, we impose certain syntactic *safety* restrictions on temporal queries that guarantee decidability.

Intuitively, a temporal logic formula is *temporally safe*, if the time domain on which the formula is evaluated is uniformly bounded for all values of all the variables appearing in the

formula. In other words, it is guaranteed for a safe temporal logic formula that the formula is always evaluated in finite time.

Formally, a temporal logic formula is temporally safe if all the terms in its conjunctive normal form have the form $\circ^k \Box_T \phi$ or $\circ^k \Diamond_T \phi$, where $k \geq 0$ and ϕ is *any* temporal logic formula. For example, the formulas $\Diamond_{20} \Box A$, $\Box_{40} A \wedge \Diamond_{60} B$, and $\circ \Box_T (\Diamond A \vee \Box B)$ are temporally safe, and the formulas $\Diamond \Box_T A$, $\Box A \wedge \Diamond_T B$ are not temporally safe.

Besides temporal safety, we also need *structural safety* that guarantees that the temporal logic query returns only finite answers. Structural safety is defined as in [Ull88] with the additional provision that the temporal operators of bounded and unbounded necessity and possibility, next, and their mirror images produce structurally safe formulas if their operands are structurally safe.

A temporal logic formula is *safe* if it is both structurally and temporally safe. For example, queries **Q1** – **Q3** are safe.

5 Summary

We have presented the language PTL suitable for programming reactive systems. The language is based on a fragment of predicate temporal logic with inflationary semantics. The formulas in the language have clausal form, support negations both in the head and in the body of a rule, support temporal operators of bounded necessity and possibility, and also support non-deterministic variables. PTL satisfies the requirements for programming reactive systems as presented in this paper. Finally, PTL supports temporal queries that can be interpreted as calls to PTL programs.

References

[AM89] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8:277–295, 1989.

[AV88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proceedings of PODS Symposium*, pages 240–250, 1988.

[Bau89] M. Baudinet. Temporal logic programming is complete and expressive. In *Symp. on Principles of Programming Languages*, pages 267–280, 1989.

[BFG+89] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A framework for programming in temporal logic. In *Stepwise Refinement of Distributed Systems*, pages 94–129. Springer-Verlag, 1989. LNCS 430.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972. New York.

[FKTMo86] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Tokio: Logic programming language based on temporal logic and its compilation to Prolog. In *Third International Conference on Logic Programming*, pages 695–709. Springer-Verlag, 1986. LNCS 225.

[Gab89] D. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of Colloquium on Temporal Logic in Specification*, pages 402–450. Springer-Verlag, 1989. LNCS 398.

[Gro87] Mikell P. Groover. *Automation, Production Systems, and Computer Integrated Manufacturing.* Prentice-Hall, 1987. Chapter 17.

[GS86] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.

[KKN+90] D. Kato, T. Kikuchi, R. Nakajima, J. Sawada, and H. Tsuiki. Modal logic programming. In *VDM and Z - Formal Methods in Software Development.* Springer-Verlag, 1990. LNCS 428.

[KP88] P. G. Kolaitis and C. H. Papadimitriou. Why not negation by fixpoint? In *Proceedings of PODS Symposium*, pages 231–239, 1988.

[Kro87] Fred Kroger. *Temporal Logic of Programs.* Springer-Verlag, 1987. EATCS Monographs on Theoretical Computer Science.

[MH69] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificail intelligence. In *Machine Intelligence 4.* Edinburgh University Press, 1969.

[Mos86] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, England, 1986.