From Safety Properties to Temporal Logic Programming: A Study in Fixed Point Temporal Logic

Stephan Merz
Institut für Informatik, Ludwig-Maximilians-Universität
Theresienstr. 39, W-8000 München 2, F.R.G.
merz@informatik.uni-muenchen.de

Abstract

Starting from a syntactic characterization of safety properties of reactive systems in the fixed point temporal logic νTL, we define a class of formulas for which a model can be effectively generated and which can thus be considered as the core of a programming language based on temporal logic.

1 Introduction

We distinguish between two kinds of computer programs: *transformational programs* compute some result(s) from initially given input data and then terminate. Such programs can therefore be modelled as mathematical functions or relations between suitable data domains. Consequently, specification techniques for transformational programs are usually based on classical first-order logic. On the other hand, *reactive programs* such as operating systems, database servers etc. continually accept requests from their environment (either the user or other software modules) and react to those requests, thereby changing state (e.g., storing data tuples in a database or deleting previously stored items) which affects the reactions to future requests. In general, such systems are not intended to terminate at any time. The behaviour of reactive systems is best explained in terms of runs, i.e. conceptually infinite sequences of states (or state transitions). It therefore appears advantageous to adopt mathematical and logical formalisms for the specification of reactive systems which are explicitly based on such notions. One such formalism which has become well-established in this field is temporal logic of linear time, cf. [8, 16, 12].

Starting from an initial temporal logic specification of the software module to be developed, formal program development proceeds by *refinement*, i.e. stating more and more detailed intermediate specifications until a program can be "read off" the final specification. At every refinement step it has to be ensured that the new specification $\mathcal{S}'$ implements the previous specification $\mathcal{S}$, i.e. that every program allowed by $\mathcal{S}'$ is also allowed by $\mathcal{S}$. In a temporal logic-based formalism, this amounts to proving that every model of $\mathcal{S}'$ is also a model of $\mathcal{S}$, i.e. that $\mathcal{S}'$ implies $\mathcal{S}$. Similarly, the resulting program has to verified w.r.t. the final specification, usually by defining a temporal logic-based semantics of the programming language which has been employed. (This approach is demonstrated in [8]).

(1)	$\Box\forall job, q(Print(job) \wedge PrintQueue = q \rightarrow \bigcirc PrintQueue = delete(q, job))$
(2)	$\Box\forall job(job = head(PrintQueue) \rightarrow Print(job)\ \mathsf{atnext}\ \neg PrinterBusy)$
(3)	$\Box\forall job(Received(job) \rightarrow \Diamond Print(job))$
(4)	$\Box(IsEmpty(PrintQueue)\ \mathsf{before}\ SystemShutDown)$

Figure 1: Sample specification of a print server (excerpt)

Usually, one finds that some formulas of a specification are so detailed that they define exactly which actions to perform by the program in a given situation and could therefore be executed by an interpreter for temporal logic, whereas for other formulas it is necessary to find refinements before it is clear how to implement them. Suppose, for example, that we wanted to specify the operation of a print server which accepts print jobs from client modules, stores them in a print queue and sends them to the printer according to some servicing strategy whenever the printer is ready to accept a new job. Figure 1 displays some formulas which might occur in such a specification. (The exact semantics of the temporal operators will be given below.) The first formula should easily be implementable in some suitable programming language by including a statement such as

$$PrintQueue := Delete(PrintQueue, job)$$

in the code of the procedure *Print*. The second formula does not specify an immediate state transition but schedules one for the next time when the printer becomes ready. This behaviour may be implemented (assuming that the head of the queue remains unchanged between sending jobs to the printer) by monitoring the *PrinterBusy* condition (which in a real system would be a signal issued by the printer) and performing the required action at the next time the condition is false. In contrast, it is not as straightforward to implement the third formula which requires each print job to be *eventually* sent to the printer: There exist a number of different strategies how to schedule print jobs which satisfy this requirement, and it is hard to imagine a computer program choosing a suitable implementation without further guidance by the developer. As far as the fourth formula is concerned, it appears not to be realizable by the print server module: Assuming that *SystemShutDown* is a condition which is generated externally to the print server (e.g., in response to a user-issued command), it cannot be influenced by the print server. To ensure that every run of the system (seen as a whole) obeys the requirement expressed by this formula, the module responsible for shutting down the system has to check if there still are jobs in the print queue and wait until the queue becomes empty.

To gain a better understanding of the differences between formulas such as those of figure 1, we will in the following study semantic properties of certain classes of temporal logic formulas. To make our results as widely applicable as possible and to obtain a uniform technical framework, we will introduce temporal operators using fixed point definitions. It turns out that all temporal connectives defined in a more conventional way, such as those used in [8] or [12], can be defined in this way.

The organization of this paper is as follows: Section 2 gives basic results on important classes of properties independent of logical considerations. Section 3 introduces the propositional version of the fixed point temporal logic νTL. Some connections between the notions introduced that far are given in section 4, characterizing νTL formulas expressing safety properties. Section 5 discusses deterministic safety properties and the syntactic form of νTL formulas expressing them. This class of formulas will be further restricted in section 6 to yield executable temporal logic formulas, and the basic operation of an interpreter constructing models for temporal logic programs is outlined. Finally, we hint at an extension of this framework to first-order νTL and discuss our results in section 7.

2 Safety and Liveness Properties

Before introducing our temporal logic formalism, let us review some generalities on properties of reactive systems as described in [1] or [11]. Let Σ be a set (of states which may be assumed by some reactive system). Runs of the system may be represented as an infinite sequence $\sigma = (s_0, s_1, \ldots) \in \Sigma^\omega$. A ($\Sigma$-)*property* P is simply a set $P \subseteq \Sigma^\omega$ (of runs satisfying the property). For a run σ and $i \geq 0$, let $\sigma[i]$ denote the i-th element s_i of σ, and $\sigma[..i]$ denote the finite sequence $(s_0, \ldots, s_i) \in \Sigma^*$. For $\sigma \in \Sigma^*$ and $\tau \in \Sigma^* \cup \Sigma^\omega$, let $\sigma\tau$ denote the sequence obtained by concatenating σ and τ. For a property $P \subseteq \Sigma^\omega$, $\mathit{Pref}(P) = \{\sigma \in \Sigma^* \mid \exists \sigma' \in P : \sigma \prec \sigma'\}$ denotes the set of finite prefixes of runs in P.

Usually, two basic classes of properties are distinguished as in the following

Definition 2.1

- *A property* $P \subseteq \Sigma^\omega$ *is a* safety property *iff for every* $\sigma \in \Sigma^\omega$,

$$\sigma \in P \iff \forall i \geq 0 \exists \tau \in \Sigma^\omega (\sigma[..i]\tau \in P).$$

- *A property* $P \subseteq \Sigma^\omega$ *is a* liveness property *iff*

$$\forall \sigma \in \Sigma^* \exists \tau \in \Sigma^\omega (\sigma\tau \in P).$$

Informally, a property P is a safety property if for every run $\sigma \notin P$ there is a finite prefix $\sigma[..i]$ which is not a prefix of any sequence $\rho \in P$. Put differently, a violation of a safety property can be detected after a finite number of steps, i.e. a safety property, in Lamport's popular characterization [9], states that "something bad never happens". In contrast, if P is a liveness property, every finite state sequence can be extended to yield a run in P. Hence, violations of liveness properties can only be detected by viewing the run as a whole. Consequently, proof methods for safety and liveness properties are entirely different: safety properties are normally proven by induction on the length of finite prefixes of runs, whereas the proof of liveness properties usually involves appealing to well-founded orderings. This observation is embodied in proof rules of typical formal systems for program verification such as [5, 6, 15].

It is easy to prove the following fact about safety properties:

Fact 2.2

(i) *Let* $\{P_i \mid i \in I\}$ *be a set of safety properties. Then* $\bigcap_{i\in I} P_i$ *is a safety property.*

(ii) *Let* $P_1, \ldots, P_n$ *be safety properties. Then* $\bigcup_{i=1}^{n} P_i$ *is a safety property.*

(iii) Σ^ω *is the only property which is both a safety and a liveness property.*

For a liveness property P, it is obvious from the definition that every property $Q \supseteq P$ is also a liveness property. In particular, unions of liveness properties are liveness properties. However, in general even finite intersections of liveness properties do not yield liveness properties. By fact 2.2, for every property P there is a smallest safety property $S \supseteq P$ which contains P. It may easily be shown that S may be defined in two equivalent ways:

Definition 2.3 *Let* P *be any property. The* safety closure $\mathit{safe}(P)$ *of* P *is the smallest safety property containing* P*,*

$$\begin{aligned} \mathit{safe}(P) &= \bigcap\{S \subseteq \Sigma^\omega \mid S \text{ is a safety property and } P \subseteq S\} \\ &= \{\sigma \in \Sigma^\omega \mid \forall \sigma' \in \Sigma^+ \, (\sigma' \prec \sigma \Rightarrow \sigma' \in \mathit{Pref}(P))\}. \end{aligned}$$

One may also associate a liveness property with a given property P:

Definition 2.4 *Let* P *be any property. Then*

$$\mathit{live}(P) := P \cup (\Sigma^\omega \setminus \mathit{safe}(P))$$

is called the liveness extension *of* P*.*

The notions introduced above are related by the following theorem:

Theorem 2.5 *For every property* P*,* $\mathit{live}(P)$ *is a liveness property. Moreover,* $\mathit{safe}(P) \cap \mathit{live}(P) = P$*, and* $\mathit{safe}(P) \cup \mathit{live}(P) = \Sigma^\omega$*.*

Proof: To prove that $\mathit{live}(P)$ is indeed a liveness property, let $\sigma \in \Sigma^*$ be any finite state sequence. If $\sigma \in \mathit{Pref}(P)$, then by definition there is some $\tau \in \Sigma^\omega$ such that $\sigma\tau \in P \subseteq \mathit{live}(P)$. If $\sigma \notin \mathit{Pref}(P)$, then for every $\tau \in \Sigma^\omega$, $\sigma\tau \notin \mathit{safe}(P)$ (by the second definition of $\mathit{safe}(P)$). In particular, there is some $\tau \in \Sigma^\omega$ such that $\sigma\tau \notin \mathit{safe}(P)$, but this implies $\sigma\tau \in \mathit{live}(P)$.

The asserted equations follow from simple calculations:

$\mathit{safe}(P) \cap \mathit{live}(P) = \mathit{safe}(P) \cap (P \cup (\Sigma^\omega \setminus \mathit{safe}(P))) = P$

$\mathit{safe}(P) \cup \mathit{live}(P) = \mathit{safe}(P) \cup (P \cup (\Sigma^\omega \setminus \mathit{safe}(P))) = \Sigma^\omega$. ♠

In particular, therefore, every property can be represented as the intersection of a safety property and a liveness property.

Example 2.6 Let $A, B \subseteq \Sigma$ and let U be the property defined by

$$\begin{array}{rcl} U &=& \{\sigma \in \Sigma^\omega \mid \exists i \geq 0 : \sigma[i] \in B \text{ and } \forall j < i : \sigma[j] \in A\}. \\ \text{Then, } \mathit{Pref}(U) &=& \{\sigma \in \Sigma^* \mid \exists 0 \leq i < |\sigma| : \sigma[i] \in B \text{ and } \forall j < i : \sigma[j] \in A\} \cup \\ && \{\sigma \in \Sigma^* \mid \forall 0 \leq i < |\sigma| : \sigma[i] \in A\} \\ \text{and } \mathit{safe}(U) &=& \{\sigma \in \Sigma^\omega \mid \sigma \in U \text{ or } \forall i \geq 0 : \sigma[i] \in A\}. \\ \text{Hence, } \mathit{live}(U) &=& U \cup \{\sigma \in \Sigma^\omega \mid \sigma \notin U \text{ and } \exists i \geq 0 : \sigma[i] \notin A\} \\ &=& \{\sigma \in \Sigma^\omega \mid \exists i \geq 0 : \sigma[i] \in B \text{ or } \sigma[i] \notin A\} \end{array}$$

Notice that the property $L = \{\sigma \in \Sigma^\omega \mid \exists i \geq 0\ \sigma[i] \in B\}$ is also a liveness property satisfying $\mathit{safe}(U) \cap L = U$, and that $L \subseteq \mathit{live}(U)$ (the inclusion is strict if $\Sigma \backslash (A \cup B) \neq \emptyset$). Thus, $\mathit{live}(P)$ need not be the smallest liveness property which contains P and satisfies the above equation. Since intersections of liveness properties do not necessarily yield liveness properties, such a property need in general not exist. ♠

3 Propositional νTL

Let us now define the temporal logic νTL introduced in [2] whose semantics is based on fixed point definitions. νTL is closely related to the μ-calculus studied in [7].

Let $\mathcal{V} = \{p, q, \ldots\}$ and $\mathcal{X} = \{X, Y, \ldots\}$ be denumerable non-empty sets of *proposition letters* and *proposition variables*, respectively. Formulas of propositional νTL are built from $\mathcal{V}$ and $\mathcal{X}$ by applying the boolean connectives $\neg$ and $\wedge$ and the unary temporal operator $\circ$ (*read*: next). Moreover, for every formula F and every proposition variable $X \in \mathcal{X}$ all of whose free occurrences in F are positive, i.e. in the scope of an even number of negation symbols, $(\nu X\ F)$ is a formula. The occurrence of a variable X in a formula F is called *bound* if it is contained in some subformula $(\nu X\ G)$ of F. Otherwise, it is called *free*. A formula containing no free (occurrences of) proposition variables is called a *closed formula*.

If F and G are formulas and X is a proposition variable, $F[G/X]$ denotes the formula resulting from F by replacing all free occurrences of X by G, renaming bound proposition variables in F if necessary to avoid free occurrences of proposition variables in G to become bound in $F[G/X]$. Additional boolean connectives like $\vee$, $\rightarrow$ and $\leftrightarrow$ are defined as the usual abbreviations and will be used liberally. Moreover, the following fixed point and temporal operators are defined as abbreviations:

$$\begin{array}{rcll} (\mu X\ F) &\equiv& \neg(\nu X\ \neg F[\neg X/X]) & \text{if all free occurrences of } X \text{ in } F \text{ are positive} \\ \Box F &\equiv& \nu X\ (F \wedge \circ X) & (\mathit{read}\text{: always } F) \\ \Diamond F &\equiv& \mu X\ (F \vee \circ X) & (\mathit{read}\text{: eventually } F) \\ F \ \mathsf{atnext}\ G &\equiv& \nu X\ ((G \rightarrow F) \wedge (\neg G \rightarrow \circ X)) & \\ F \ \mathsf{unless}\ G &\equiv& \nu X\ (\neg G \rightarrow (F \wedge \circ X)) & \end{array}$$

$$\begin{array}{rcl} F \ \mathsf{until}\ G & \equiv & \mu X\,(\neg G \rightarrow (F \wedge \circ X)) \\ F \ \mathsf{before}\ G & \equiv & \nu X\,(\neg F \rightarrow (\neg G \wedge \circ X)) \\ F \ \mathsf{upto}\ G & \equiv & \nu X\,(F \wedge (\neg G \rightarrow \circ X)) \end{array}$$

A *temporal structure* σ for propositional νTL is an ω-sequence $\sigma = (s_0, s_1, \ldots)$ of states where each state is a mapping $s_i : \mathcal{V} \rightarrow \{tt, ff\}$. The semantics of propositional νTL is defined by a temporal structure σ together with a valuation $\xi : \mathcal{X} \rightarrow 2^{\mathbb{N}}$ of the proposition variables as follows:

$$\begin{array}{rcl} \sigma^\xi(v) & = & \{i \in \mathbb{N} \mid \sigma_i(v) = tt\} \quad \text{for every proposition letter } v \in \mathcal{V}, \\ \sigma^\xi(X) & = & \xi(X) \quad \text{for every proposition variable } X \in \mathcal{X}, \\ \sigma^\xi(\neg F) & = & \mathbb{N} \setminus \sigma^\xi(F), \\ \sigma^\xi(F \wedge G) & = & \sigma^\xi(F) \cap \sigma^\xi(G), \\ \sigma^\xi(\circ F) & = & \{i \in \mathbb{N} \mid i+1 \in \sigma^\xi(F)\}, \\ \sigma^\xi(\nu X\, F) & = & \bigcup\{I \subseteq \mathbb{N} \mid I \subseteq \sigma^{\xi[I/X]}(F)\}. \end{array}$$

It follows that the semantics of the μ-operator is given by

$$\sigma^\xi(\mu X\, F) \quad = \quad \bigcap\{I \subseteq \mathbb{N} \mid I \supseteq \sigma^{\xi[I/X]}(F)\}.$$

Here $\xi[I/X]$ denotes the valuation defined by $\xi[I/X](Y) = \xi(Y)$ for every variable $Y \neq X$, and $\xi[I/X](X) = I$. Intuitively, $\sigma^\xi(F)$ is the set of (indexes of) states at which F holds in the structure σ under the valuation ξ. The condition $i \in \sigma^\xi(F)$ will often be written as $\sigma_i^\xi \models F$. F is called *true in* σ *under* ξ if $\sigma_0^\xi \models F$. For a closed formula F, the definition of $\sigma^\xi(F)$ is independent of ξ which will then frequently be omitted. A closed formula is called *satisfiable* if there is a structure σ such that $\sigma \models F$. It is called *valid* if $\sigma \models F$ holds for every structure σ. A structure σ such that $\sigma \models F$ is called a *model* of F, and the set of all models of F is denoted by $Mod(F)$. A closed formula F *follows from* a set $\mathcal{F}$ of closed formulas, denoted by $\mathcal{F} \models F$, if every model of all formulas in $\mathcal{F}$ is also a model of F. Two formulas F and G are called *equivalent* if $\sigma^\xi(F) = \sigma^\xi(G)$ for every structure σ and valuation ξ.

It is straightforward to see that the above definition yields the expected results for the derived temporal operators introduced above. For example, $\sigma^\xi(\Box F) = \{i \in \mathbb{N} \mid \forall j \geq i : \sigma_j^\xi \models F\}$.

The polarity condition on free occurrences of X in F placed on formulas $(\nu X\, F)$ and $(\mu X\, F)$ ensures that the mapping $\tau_F : 2^{\mathbb{N}} \rightarrow 2^{\mathbb{N}}$ defined by $\tau_F(I) = \sigma^{\xi[I/X]}(F)$ (for fixed σ and ξ) is monotonic on the complete lattice $(2^{\mathbb{N}}, \subseteq)$. The fixpoint theorem of Knaster and Tarski [18] therefore ensures the existence of both a smallest fixed point $\mu_F \subseteq \mathbb{N}$ and a greatest fixed point $\nu_F \subseteq \mathbb{N}$ of τ_F which coincide with the definitions of $\sigma^\xi(\mu X\, F)$ and $\sigma^\xi(\nu X\, F)$, respectively. Hence, we arrive at the following theorem:

Theorem 3.1 *Let F and G be formulas, σ be a temporal structure and ξ be a valuation of the proposition variables. Then,*

(i) $\sigma^\xi(\alpha X\, F) = \sigma^\xi(F[(\alpha X\, F)/X])$ *for $\alpha \equiv \nu$ and $\alpha \equiv \mu$,*

(ii) $$\begin{array}{rcl} \sigma^\xi(G) \subseteq \sigma^\xi(F[G/X]) & \Longrightarrow & \sigma^\xi(G) \subseteq \sigma^\xi(\nu X\, F) \\ \sigma^\xi(F[G/X]) \subseteq \sigma^\xi(G) & \Longrightarrow & \sigma^\xi(\mu X\, F) \subseteq \sigma^\xi(G) \end{array}$$

Let us conclude this overview on νTL by defining three normal forms for closed νTL formulas. First, by "pulling negations inside", positive normal form is obtained where negations are only allowed to appear before proposition letters. Formally, the definition is as follows:

Definition 3.2

- *A* literal *is either of the form p or $\neg p$ for a proposition letter $p \in \mathcal{V}$, or a constant tt or ff.*
- *A* formula in positive normal form *(PNF formula) is built from literals and proposition variables $X \in \mathcal{X}$ using the operators $\wedge$, $\vee$, $\circ$, ν and μ (but not $\neg$).*

Observe that the polarity condition for $(\nu X\, F)$ and $(\mu X\, F)$ is trivially satisfied for PNF formulas.

Our next normal form eliminates bound occurrences of proposition variables which are not "guarded" by a $\circ$-operator:

Definition 3.3 *A PNF formula F is in* guarded normal form *(a GNF formula) if every free occurrence of X in subformulas of the form $(\alpha X\ G)$ for $\alpha \in \{\mu, \nu\}$ is in the scope of at least one $\circ$-operator.*

For example, $\nu X\, (p \vee \circ(q \wedge X))$ is a GNF formula whereas $p \vee \circ \nu X\, (q \wedge X)$ is not. The following theorem on guarded normal form is proven in [14]:

Theorem 3.4 *For every PNF formula F, an equivalent GNF formula can be obtained by replacing every free occurrence of X in subformulas $(\alpha X\ G)$ of F which is not in the scope of a $\circ$-operator by tt if $\alpha \equiv \nu$, and by ff if $\alpha \equiv \mu$.*

Finally, we will define a disjunctive normal form for propositional νTL:

Definition 3.5 *A GNF formula F is in* disjunctive normal form *(a DNF formula) if F is of the form $C_1 \vee \ldots C_m$ for some $m \geq 0$ (where the empty disjunction represents ff), and every C_i is of the form $L_i^1 \wedge \ldots \wedge L_i^{n_i}$ for some $n_i > 0$ such that every L_i^j is either*

- *$\circ^k p$ or $\circ^k \neg p$ for some proposition letter $p \in \mathcal{V}$ and some $k \geq 0$,*
- *$\circ^k X$ for some proposition variable $X \in \mathcal{X}$ and some $k \geq 0$, or*
- *$\circ^k(\alpha X\ B)$ where B is a DNF formula, $\alpha \in \{\mu, \nu\}$, and $k \geq 0$.*

The construction of an equivalent DNF formula G for a given GNF formula F proceeds similarly as in classical logic by applying the equivalences listed below to every subformula of F to which they apply:

$$\begin{array}{rclrcl}
\circ(A \wedge B) & \equiv & \circ A \wedge \circ B & \circ(A \vee B) & \equiv & \circ A \vee \circ B \\
A \wedge (B \vee C) & \equiv & (A \wedge B) \vee (A \wedge C) & & & \\
\circ tt & \equiv & tt & \circ ff & \equiv & ff \\
A \wedge tt & \equiv & A & A \wedge ff & \equiv & ff \\
A \vee tt & \equiv & tt & A \vee ff & \equiv & A
\end{array}$$

4 Characterizing safety properties in νTL

Denoting by Σ the set of assignments $\sigma : \mathcal{V} \to \{tt, ff\}$ (for some fixed set $\mathcal{V}$ of proposition letters), we may apply the notions of section 2 and consider the set $Mod(F) \subseteq \Sigma^\omega$ of models of some closed νTL formula F as the property defined by F. In particular, we may ask if there are syntactic criteria for formulas expressing safety or liveness properties. The following theorem provides a partial answer to this question:

Theorem 4.1 *Let F be a closed PNF formula which does not contain subformulas of the form $(\mu X\ G)$. Then $Mod(F)$ is a safety property.*

Proof: We will show the following more general assertion: Let F be a (not necessarily closed) PNF formula without any subformula $(\mu X\ G)$ and $\sigma \in \Sigma^\omega$ be a temporal structure. We show that for every set $I \subseteq I\!N$ and all valuations ξ of the proposition variables, it holds that:

$$(*) \quad (\forall i \geq 0 \exists \tau \in \Sigma^\omega : I \subseteq (\sigma[..i]\tau)^\xi(F)) \implies I \subseteq \sigma^\xi(F)$$

The claim of the theorem follows from $(*)$ by choosing $I = \{0\}$ and the assumption that F is closed. The proof of the auxiliary assertion runs by induction on F (assuming the antecedent of $(*)$):

- F is a literal or a proposition variable: By the assumption, for every $i \in I$ there is some $\tau \in \Sigma^\omega$ such that $i \in (\sigma[..i]\tau)^\xi(F)$, implying $i \in \sigma^\xi(F)$ by definition. Hence, $I \subseteq \sigma^\xi(F)$.
- $F \equiv G \wedge H$: For every $i \geq 0$ there is some $\tau \in \Sigma^\omega$ such that $I \subseteq (\sigma[..i]\tau)^\xi(G \wedge H) = (\sigma[..i]\tau)^\xi(G) \cap (\sigma[..i]\tau)^\xi(H)$. By the induction hypothesis for G and H, $I \subseteq \sigma^\xi(G) \cap \sigma^\xi(H) = \sigma^\xi(G \wedge H)$.
- $F \equiv G \vee H$: Assume that $I \not\subseteq \sigma^\xi(G \vee H)$. Then $I \not\subseteq \sigma^\xi(G)$ and $I \not\subseteq \sigma^\xi(H)$. By the induction hypothesis there are $j, k \geq 0$ such that for all $\tau \in \Sigma^\omega$, $I \not\subseteq (\sigma[..j]\tau)^\xi(G)$ and $I \not\subseteq (\sigma[..k]\tau)^\xi(H)$. Hence, there is some $l \geq 0$ (viz., $l = \max(j,k)$) such that for all $\tau \in \Sigma^\omega$, $I \not\subseteq (\sigma[..l]\tau)^\xi(G \vee H)$.
- $F \equiv \circ G$: For every $i \geq 0$ there is some $\tau \in \Sigma^\omega$ such that $I \subseteq (\sigma[..i]\tau)^\xi(\circ G) = \{j \mid j+1 \in (\sigma[..i]\tau)^\xi(G)\}$. Let I^{+1} denote the set $\{i+1 \mid i \in I\}$, then for every $i \geq 0$ there is $\tau \in \Sigma^\omega$ such that $I^{+1} \subseteq (\sigma[..i]\tau)^\xi(G)$, hence, applying the induction hypothesis for I^{+1}, it follows that $I^{+1} \subseteq \sigma^\xi(G)$, i.e. $I \subseteq \sigma^\xi(\circ G)$.
- $F \equiv (\nu X G)$: For every $i \geq 0$ there is some $\tau \in \Sigma^\omega$ such that $I \subseteq (\sigma[..i]\tau)^\xi(\nu X G) = \bigcup\{J \subseteq I\!N \mid J \subseteq (\sigma[..i]\tau)^{\xi[J/X]}(G)\}$. Then for every $k \in I$ there is some set $J_k \subseteq I\!N$ such that $k \in J_k$ and $J_k \subseteq (\sigma[..i]\tau)^{\xi[J_k/X]}(G)$, hence by monotonicity $J_k \subseteq (\sigma[..i]\tau)^{\xi[(\bigcup_{k\in I} J_k)/X]}(G)$, and therefore $I \subseteq \bigcup_{k\in I} J_k \subseteq (\sigma[..i]\tau)^{\xi[(\bigcup_{k\in I} J_k)/X]}(G)$. Denoting $\bigcup_{k\in I} J_k$ by M_i, this means that for every $i \geq 0$ there is some $\tau \in \Sigma^\omega$ and some $M_i \subseteq I\!N$ such that $I \subseteq M_i$ and $M_i \subseteq (\sigma[..i]\tau)^{\xi[M_i/X]}(G)$. One may even choose $M_i = M_{i+1}$ for every $i \geq 0$: Assuming that there is some $\tau \in \Sigma^\omega$ such that

$I \subseteq M_{i+1}$ and $M_{i+1} \subseteq (\sigma[..i+1]\tau)^{\xi[M_{i+1}/X]}(G)$, it follows that there is $\tau' \in \Sigma^\omega$ such that $M_{i+1} \subseteq (\sigma[..i]\tau')^{\xi[M_{i+1}/X]}(G)$ (choose $\tau' := \sigma[i+1]\tau$). But then, there is some $M \subseteq I\!N$ such that $I \subseteq M$ and for every $i \geq 0$ there is some $\tau \in \Sigma^\omega$ such that $M \subseteq (\sigma[..i]\tau)^{\xi[M/X]}(G)$. Applying the induction hypothesis for M and $\xi[M/X]$ yields $M \subseteq \sigma^{\xi[M/X]}(G)$, hence $I \subseteq M \subseteq \bigcup\{J \subseteq I\!N \mid J \subseteq \sigma^{\xi[J/X]}(G)\} = \sigma^{\eta,\xi}(\nu X\, G)$.

♠

Remark: The proof is easily extended to a first-order version of νTL if no subformula $(\nu X\ G)$ of F appears in the scope of an $\exists$ quantifier (cf. [14]). On the other hand, examples can be given for properties expressed by first-order νTL formulas using only greatest fixed points but without this additional proviso which do not express a safety property (consider $\exists n \Box z \leq n$ where z is a flexible variable interpreted over the natural numbers). This observation is consistent with the claim of fact 2.2 that in general, only finite unions of safety properties yield safety properties.

Let us now address the converse statement: Can every safety property on Σ^ω be expressed by a PNF formula without smallest fixed points? We have to be a bit careful because, clearly, there are properties which are not expressible in νTL[1] such as the property that p holds at every state $\sigma[p]$ where p is a prime number.

In the remainder of this section, we will employ yet another normal form of νTL formulas which, however, is not defined in purely syntactic terms: Let $F(\alpha X\, G)$ be some GNF formula and $(\alpha X\ G)$ be a fixed point subformula of F some occurrences of which are not in the scope of a $\bigcirc$-operator. By "unwinding" (theorem 3.1), F is equivalent to $F(G[(\alpha X\ G)/X])$. Since F, hence also G, were assumed to be GNF formulas, every occurrence of $(\alpha X\, G)$ in the transformed formula will be in the scope of at least one $\bigcirc$-operator. Proceeding inductively and transforming the resulting formula to disjunctive normal form, for every $n \geq 0$ there is some DNF formula $\mathcal{U}^n(F) \equiv \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} L_{ij}$ (the *n-th unfolding of F*) which is equivalent to F such that all fixed point subformulas of F are in the scope of at least n $\bigcirc$-operators. Call the formula $\mathcal{R}^n(F)$ which results from $\mathcal{U}^n(F)$ by deleting all L_{ij} which are in the scope of less than n $\bigcirc$-operators and eliminating empty conjunctions the *n-th reduced unfolding of F*. Call F a *reduced formula* if either $F \equiv \mathit{ff}$ or for every $n \geq 0$, every disjunct of $\mathcal{R}^n(F)$ is satisfiable.

Example 4.2 Consider the formula $F \equiv \Diamond p \wedge \bigcirc\bigcirc\bigcirc\Box\neg p$. Then, e.g., $\mathcal{U}^3(F) \equiv (p \wedge \bigcirc\bigcirc\bigcirc\Box\neg p) \vee (\bigcirc p \wedge \bigcirc\bigcirc\bigcirc\Box\neg p) \vee (\bigcirc\bigcirc p \wedge \bigcirc\bigcirc\bigcirc\Box\neg p) \vee (\bigcirc\bigcirc\bigcirc\Diamond p \wedge \bigcirc\bigcirc\bigcirc\Box\neg p)$, and $\mathcal{R}^3(F) \equiv \bigcirc\bigcirc\bigcirc\Box\neg p \vee \bigcirc\bigcirc\bigcirc\Box\neg p \vee \bigcirc\bigcirc\bigcirc\Box\neg p \vee (\bigcirc\bigcirc\bigcirc\Diamond p \wedge \bigcirc\bigcirc\bigcirc\Box\neg p)$. $\mathcal{R}^3(F)$ contains the unsatisfiable disjunct $\bigcirc\bigcirc\bigcirc\Diamond p \wedge \bigcirc\bigcirc\bigcirc\Box\neg p$. Hence F is not reduced. A reduced formula which is equivalent to F is $F' \equiv (p \vee \bigcirc p \vee \bigcirc\bigcirc p) \wedge \bigcirc\bigcirc\bigcirc\Box\neg p$. ♠

[1]It has been shown in [19, 17] that propositional νTL is expressively complete w.r.t. the monadic second-order theory $S1S$ of linear orders.

Lemma 4.3 *For every propositional νTL formula F there is an equivalent reduced formula F'.*

Proof: For any formula A, let $|A|_n$ denote the number of disjuncts of $\mathcal{U}^n(A)$. Assume that for some formula F there were no equivalent reduced formula F'. In particular, for some smallest $n \geq 0$, $\mathcal{R}^n(F)$ contains an unsatisfiable disjunct $D' \equiv \bigwedge_{j=1}^{k} L_j$. Let F be chosen such that for every formula A such that $|A|_n < |F|_n$ there is an equivalent reduced formula A'. Now, $\mathcal{U}^n(F)$ contains a disjunct $D \equiv \bigwedge_{j=1}^{m} L_j$ from which D' was obtained by deleting those L_j which are in the scope of less than n $\circ$-operators. Clearly, D is also unsatisfiable. Hence, the formula F'' obtained from $\mathcal{U}^n(F)$ by deleting D is equivalent to $\mathcal{U}^n(F)$, hence also to F. But all subformulas $(\alpha X\, G)$ of F'' appear in the scope of at least n $\circ$-operators by definition. Hence, $\mathcal{U}^n(F'') \equiv F''$, and $|F''|_n < |F|_n$. By the choice of F there is some reduced formula F' which is equivalent to F'', hence also to F, and contradiction is reached. ♠

Let F be a PNF formula, and denote by F^ν the formula resulting from F by replacing all μ-operators in F by ν. Not surprisingly, every model of F is a model of F^ν:

Lemma 4.4 *Let F be a closed PNF formula. Then $Mod(F) \subseteq Mod(F^\nu)$.*

Proof: By induction on F, it is easy to show that $\sigma^\xi(F) \subseteq \sigma^\xi(F^\nu)$ holds for every PNF formula F, every temporal structure σ and valuation ξ of the proposition variables. ♠

However, for reduced formulas even the following assertion can be established:

Lemma 4.5 *Let F be a closed and reduced GNF formula and $\sigma = (s_0, \ldots, s_n) \in \Sigma^+$ be a finite sequence of states such that for every $\tau \in \Sigma^\omega$, $\sigma\tau \not\models F$. Then $\sigma\tau \not\models F^\nu$ for every $\tau \in \Sigma^\omega$.*

Proof: Consider $\mathcal{U}^{n+1}(F) \equiv \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} L_{ij}$, which by construction is also a closed formula. Since F and $\mathcal{U}^{n+1}(F)$ are equivalent formulas, it follows that for every $\tau \in \Sigma^\omega$, $\sigma\tau \not\models \mathcal{U}^{n+1}(F)$. Thus, for every $\tau \in \Sigma^\omega$ and $1 \leq i \leq m$, $\sigma\tau \not\models \bigwedge_{j=1}^{n_i} L_{ij}$. We distinguish two possible cases:

Case 1: There is some $\tau \in \Sigma^\omega$ such that for every i there is some j such that $\sigma\tau \not\models L_{ij}$ and L_{ij} has at most n leading $\circ$-operators. By definition of $\mathcal{U}^{n+1}(F)$, all those L_{ij} must be of the form $\circ^k(\neg)p$ for some $p \in \mathcal{V}$. Now consider $\mathcal{U}^{n+1}(F^\nu) \equiv \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} L_{ij}^\nu$: Since $(\circ^k(\neg)p)^\nu \equiv \circ^k(\neg)p$ for $p \in \mathcal{V}$, it follows $\sigma\tau \not\models \mathcal{U}^{n+1}(F^\nu)$, hence also $\sigma\tau \not\models F^\nu$ as asserted.

Case 2: For every $\tau \in \Sigma^\omega$ there is some $1 \leq i \leq m$ such that $\sigma\tau \models L_{ij}$ for all L_{ij} with at most n leading $\circ$-operators. Let, w.l.o.g., $\mathcal{L}_1 = \{L_{i1}, \ldots, L_{ik_i}\}$ be the set of such L_{ij}, and $\mathcal{L}_2 = \{L_{ik_i+1}, \ldots, L_{in_i}\}$ be the set of the remaining L_{ij}. As above, the L_{ij} of $\mathcal{L}_1$ are of the form $\circ^k(\neg)p$ for some $k \leq n$ and $p \in \mathcal{V}$. Choose any τ and i such that $\sigma\tau \models \bigwedge_{j=1}^{k_i} L_{ij}$. It is easy to see that $\sigma\tau \models \bigwedge_{j=1}^{k_i} L_{ij}$ for every $\tau \in \Sigma^\omega$ since the semantics of these formulas does not depend on τ. Hence, by the assumption $\sigma\tau \not\models \bigwedge_{j=1}^{n_i} L_{ij}$ for every $\tau \in \Sigma^\omega$, it follows that $\sigma\tau \not\models \bigwedge_{j=k_i+1}^{n_i} L_{ij}$ for every $\tau \in \Sigma^\omega$. Now, the truth of this formula does not depend on σ, hence $\bigwedge_{j=k_i+1}^{n_i} L_{ij}$ is unsatisfiable. But this formula is one of the disjuncts appearing in $\mathcal{R}^{n+1}(F)$, contradicting the assumption that F was reduced. Hence, this case is impossible and the lemma is shown. ♠

The preceding lemmas allow to characterize the safety closure of reduced formulas:

Theorem 4.6 *Let F be a closed and reduced νTL formula. Then $safe(Mod(F)) = Mod(F^\nu)$.*

Proof: By theorem 4.1, $Mod(F^\nu)$ is a safety property, and by lemma 4.4, $Mod(F) \subseteq Mod(F^\nu)$. Assume P were a safety property such that $Mod(F) \subseteq P$ and $Mod(F^\nu) \not\subseteq P$. Then there is some $\sigma \in Mod(F^\nu)$ such that $\sigma \notin P$. Since P is a safety property, there is some $n \in \mathbb{N}$ such that $\sigma[..n]\tau \notin P$ for every $\tau \in \Sigma^\omega$. By $Mod(F) \subseteq P$, $\sigma[..n]\tau \notin Mod(F)$ for every $\tau \in \Sigma^\omega$, which by lemma 4.5 implies that $\sigma[..n]\tau \notin Mod(F^\nu)$ for every $\tau \in \Sigma^\omega$, contradicting the assumption $\sigma \in Mod(F^\nu)$. ♠

As an immediate corollary we obtain the answer to the question posed above:

Corollary 4.7 *For every closed νTL formula F such that $Mod(F)$ is a safety property, there is an equivalent PNF formula G such that no subformula of G is of the form $(\mu X\ H)$.*

Proof: By lemma 4.3, there is a reduced formula F' which is equivalent to F, i.e. $Mod(F) = Mod(F')$. In particular, $Mod(F')$ is a safety property and by definition, $safe(Mod(F')) = Mod(F')$. On the other hand, by theorem 4.6 it follows that $safe(Mod(F')) = Mod((F')^\nu)$, hence the assertion, taking $G \equiv (F')^\nu$. ♠

5 Deterministic safety properties

In our quest to identify easily implementable or even directly executable temporal logic formulas, the characterization of safety properties studied in section 4 is only the first step: Certainly liveness properties do not help in determining which state transitions to allow for the program under development as they only constrain runs as a whole but not (finite sequences of) individual transitions. In the following, we study a more restricted class of νTL formulas which will be seen to be the basis for the definition of "executable" formulas.

Definition 5.1

- *Every literal is an antecedent. If A and B are antecedents, then $A \wedge B$ and $A \vee B$ are also antecedents.*
- *Every literal is a pre-DSF. If A is an antecedent, $X \in \mathcal{X}$ is a proposition variable and F, G are pre-DSFs, then $F \wedge G$, $\circ F$, $\circ X$, $A \rightarrow F$ and $(\nu X\ F)$ are pre-DSFs.*

- *Every closed pre-DSF is a* deterministic safety formula *(DSF).*

For example, formulas like

$$p\vee q \to r\wedge s, \quad \Box(\neg q \to \neg p) \text{ or } \Box(p \to \neg q \text{ unless } r) \equiv \nu X(p \to \nu Y(\neg r \to \neg q \wedge \bigcirc Y) \wedge \bigcirc X)$$

are deterministic safety formulas, whereas

$$r\wedge s \to p\vee q, \quad \neg q \to \Diamond\neg p \text{ or } \Box(p \to \neg q \text{ until } r) \equiv \nu X(p \to \mu Y(\neg r \to \neg q \wedge \bigcirc Y) \wedge \bigcirc Y)$$

are not. By theorem 4.1, every DSF expresses a safety property.

It is interesting to note that the class of properties definable by deterministic safety formulas coincides with that definable by *safety automata* [11] which are deterministic and complete Streett automata containing a designated failure state q_f which accept a temporal structure σ iff the (unique) run over σ does not contain q_f. Using a result shown in [11], this observation leads to the following fact whose proof is contained in [14]:

Fact 5.2 *For every property $P \subseteq \Sigma^\omega$, the following assertions are equivalent:*

(i) *P is a safety property which is recognizable by deterministic Streett automata.*

(ii) *$P = Mod(\varphi)$ for some deterministic safety formula φ.*

In the following, however, a different automata-based representation of DSFs will prove more useful:

Definition 5.3 *A* DSF automaton $T = (N, I, S, \vartheta, \zeta, n_0)$ *consists of:*

- *a finite non-empty set N of* nodes*,*
- *an acyclic relation $I \subseteq N \times N$ representing* internal edges *(i.e. $(n, n) \notin I^+$ for every $n \in N$),*
- *a relation $S \subseteq N \times N$ representing* state transitions*,*
- *a* node labelling *ϑ assigning a finite set of literals $\vartheta(n)$ to every node $n \in N$,*
- *an* edge labelling *ζ assigning an antecedent $\zeta(n, n')$ to every internal edge $(n, n') \in I$,*
- *and a designated* initial state *$n_0 \in N$.*

T is called normal *if every edge label $\zeta(n, n')$ is a disjunction $L_1 \vee \ldots \vee L_k$ of literals.*

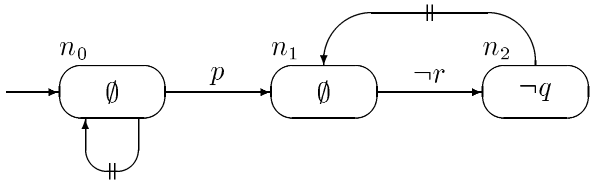


Figure 2: Graphical representation of a DSF automaton

DSF automata can be represented graphically like in figure 2: Nodes and their labels are represented by literal-inscribed ovals, internal edges are drawn as arrows inscribed with the corresponding edge label whereas "crossed" arrows are used for state transitions.

We will frequently identify a finite set $\{L_1, \ldots, L_k\}$ of literals like those appearing as node labels of transition graphs with the conjunction $L_1 \wedge \ldots \wedge L_k$, interpreting the empty set as representing *tt*.

Definition 5.4 *The* marking *of a DSF automaton $T = (N, I, S, \vartheta, \zeta, n_0)$ over a temporal structure σ is defined as the ω-sequence $M^\sigma = (m_0^\sigma, m_1^\sigma, \ldots)$ where for every $i \geq 0$, $m_i^\sigma \subseteq N$ is the smallest set of nodes satisfying the following conditions:*

- $n_0 \in m_0^\sigma$
- *If $n \in m_i^\sigma$ and $(n, n') \in I$ is an internal edge such that $\sigma_i \models \zeta(n, n')$ then $n' \in m_i^\sigma$.*
- *If $n \in m_i^\sigma$ and $(n, n') \in S$ is a state transition then $n' \in m_{i+1}^\sigma$.*

σ is accepted *by T if for every $i \in \mathbb{N}$, all labels of all nodes $n \in m_i^\sigma$ are true at $\sigma[i]$:*

$$\sigma_i \models \bigwedge_{n \in m_i^\sigma} \vartheta(n) \quad \textit{for every } i \in \mathbb{N}.$$

DSFs and DSF automata agree w.r.t. the class of temporal structures they can define. This is expressed by the two following lemmas:

Lemma 5.5 *For every DSF φ there is a DSF automaton T^φ which accepts exactly the models of φ.*

Proof (sketch): The construction of T^φ from φ is similar to the tableau construction for linear time temporal logic described in [21, 13] except that there is no need to treat eventuality formulas since φ is a safety formula. As an example, figure 2 represents the DSF automaton corresponding to the DSF $\Box(p \rightarrow \neg q \ \mathsf{unless}\ r)$. For the full proof cf. [14]. ♠

Lemma 5.6 *For every DSF automaton T there is a DSF φ^T whose models are exactly the temporal structures accepted by T.*

Proof (sketch): Let $N = \{n_0, \ldots, n_k\}$ be the set of nodes of T. φ^T will contain the proposition variables in $\mathcal{X} = \{X_0, \ldots, X_k\}$. First, define formulas ψ_j^X and δ_j^X for every subset $X \subseteq \mathcal{X}$ and every $0 \leq j \leq k$ by

$$\begin{aligned} \psi_j^X &\equiv \vartheta(n_j) \wedge \bigwedge_{(n_j,n_l)\in I} (\zeta(n_j, n_l) \to \delta_l^X) \wedge \bigwedge_{(n_j,n_l)\in S} \circ\delta_l^X \\ \delta_j^X &\equiv \left\{ \begin{array}{ll} X_j & \text{if } X_j \in X \\ \nu X_j\ \psi_j^{X \cup \{X_j\}} & \text{otherwise} \end{array} \right\}. \end{aligned}$$

Now, define $\varphi^T :\equiv \delta_0^\emptyset \equiv \nu X_0\ \psi_0^{\{X_0\}}$. By induction on $\mathcal{X} \setminus X$, it is easy to prove that every formula δ_j^X and ψ_j^X is a pre-DSF whose free variables are among X. In particular, φ^T is a deterministic safety formula. For the proof that the models of φ^T are exactly the structures accepted by T see [14]. ♠

6 Program Clauses

Deterministic safety formulas will be at the base of our definition of "executable" temporal logic, and DSF automata will be the means of giving an operational (or "imperative") semantics to DSFs. However, some additional considerations are necessary for the definition of proper program clauses. The state of a reactive system is determined by the values assumed by its state components (such as variables, signal, channels etc.) as well as those of its environment. State components will be represented by proposition letters, so we will partition $\mathcal{V}$ into two sets $\mathcal{V}_E$ and $\mathcal{V}_\Pi$ of proposition letters representing environment-owned and system-owned state components, respectively. In particular, environment components may only be "read", but not "set" by the system. Viewing DSF automata as the operational representation of program clauses, it is natural to consider literals appearing in edge labels as conditions to be *tested* (i.e., "read") whereas node labels $\vartheta(n)$ *determine* (i.e., "write") the values of literals in those states $\sigma[i]$ for which $n \in m_i^\sigma$. To obtain an efficient and backtracking-free algorithm constructing a model of a set of DSFs, mutual dependencies between literals will be eliminated in the following definition which requires the system developer to supply a partial order $<$ on the set $\mathcal{V}_\Pi$ of proposition letters representing system-owned state components. This so-called *dependency order* is extended to arbitrary literals $L_1 \equiv (\neg)p$, $L_2 \equiv (\neg)q$ by

$$L_1 < L_2 \quad \text{if} \quad (p, q \in \mathcal{V}_\Pi \text{ and } p < q) \text{ or } (p \in \mathcal{V}_E \cup \{tt, f\!f\} \text{ and } q \in \mathcal{V}_\Pi).$$

Now, the formal definition of program clauses, together with sets $read(A)$ and $write(P)$ of proposition letters whose values in the present state have to be defined before evaluating an antecedent or are being defined by a pre-clause, is as follows:

Definition 6.1

- *Every literal* $L \equiv (\neg)p$ *is an* antecedent. $read(L) = \{p\}$ *if* $p \in \mathcal{V}_E \cup \mathcal{V}_\Pi$ *and* $read(L) = \emptyset$ *if* $p \in \{tt, f\!f\}$.
- *If* A *and* B *are antecedents, then so are* $A \wedge B$ *and* $A \vee B$. $read(A \wedge B) = read(A \vee B) = read(A) \cup read(B)$.

- *For every proposition letter $p \in \mathcal{V}_\Pi$, p and $\neg p$ are* pre-PCs. $write(p) = write(\neg p) = \{p\}$.
- *If P, Q are pre-PCs then $P \wedge Q$ is a pre-PC.* $write(P \wedge Q) = write(P) \cup write(Q)$.
- *If P is a pre-PC or $P \equiv X$ is some proposition letter $X \in \mathcal{X}$, then $\bigcirc P$ is a pre-PC.* $write(\bigcirc P) = \emptyset$.
- *If P is a pre-PC and $X \in \mathcal{X}$ is a proposition variable then $(\nu X\ P)$ is a pre-PC.* $write(\nu X\ P) = write(P)$.
- *If A is an antecedent and P is a pre-PC such that $p < q$ holds for all $p \in read(A), q \in write(P)$, then $A \rightarrow P$ is a pre-PC.* $write(A \rightarrow P) = write(P)$.
- *Every closed pre-PC is a* program clause *(PC).*

In particular, every PC is a DSF. The examples given above for DSFs can also be turned into example for PCs by defining suitable dependency orders. However, the formula

$$\varphi \equiv \Box(\neg q \rightarrow \neg p) \wedge \Box(p \rightarrow \neg q \textsf{ unless } r)$$

obtained as the conjunction of two previous examples is a DSF but not a PC: From the first conjunct, it follows that $q < p$ would have to hold wheras the second conjunct requires $p < q$. Observe that the dependency order reflects purely operational constraints but has no logical meaning: The formula $\Box(p \rightarrow q) \wedge \Box(p \rightarrow \neg q \textsf{ unless } r)$ which is logically equivalent to φ is an acceptable PC (assuming that $p < q$ and $r < q$ both hold).

The interpreter algorithm attempts to construct a model for a set Φ of program clauses by calculating incrementally the markings of the normal DSF automata associated with the clauses in Φ, starting from the (known) values of proposition letters $p \in \mathcal{V}_E$ which represent environment state components and defining the values of proposition letters $p \in \mathcal{V}_\Pi$ according to the literals contained in the labels of nodes. The algorithm proceeds without backtracking, neither during the construction of one single state nor at the transition from one state to the next one. Hence, the finite state sequence computed up to some stage may immediately be output because there will not be any future changes to these values. Therefore, the algorithm is suitable for the interactive execution of reactive programs defined as a set of program clauses. Both space and time complexity of the construction of each state of the model are linear in the size of the automata representing the program which in turn is linear in the size of the input formulas provided that all antecedents are in conjunctive normal form.

Example 6.2 Let us illustrate the operation of the algorithm by an example: The program of figure 3 monitors a boolean environment-owned variable *in*, counting (modulo 2) the number of times *in* has changed from *ff* to *tt* and displaying this count in the system-owned boolean variable *out*. *lastin* is an auxiliary (internal) boolean variable containing the value of *in* at the last state. The dependency order is defined by $lastin < out$. DSF automata representing the program clauses as well as the output of the interpreter for the first 10 states of an example run are displayed as well.

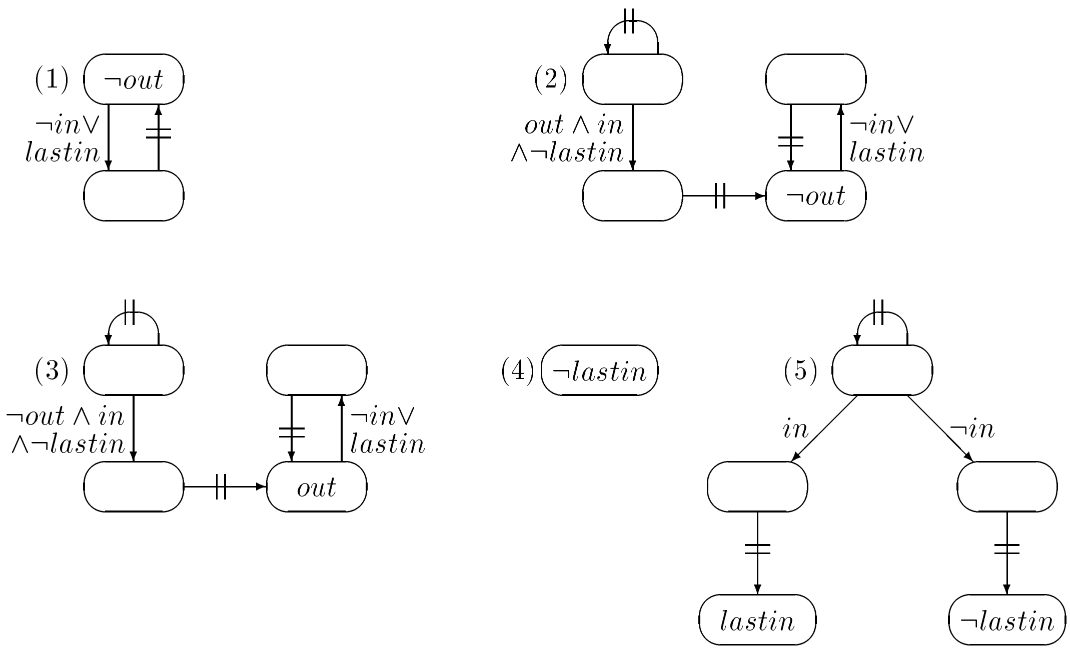
(1) $\neg out$ upto $in \wedge \neg lastin$

(2) $\Box(out \wedge in \wedge \neg lastin \rightarrow \bigcirc(\neg out$ upto $in \wedge \neg lastin))$

(3) $\Box(\neg out \wedge in \wedge \neg lastin \rightarrow \bigcirc(out$ upto $in \wedge \neg lastin))$

(4) $\neg lastin$

(5) $\Box((in \rightarrow \bigcirc lastin) \;\wedge\; (\neg in \rightarrow \bigcirc \neg lastin))$



	0	1	2	3	4	5	6	7	8	9
in	*ff*	*tt*	*tt*	*ff*	*ff*	*tt*	*ff*	*ff*	*tt*	*tt*
lastin	*ff*	*ff*	*tt*	*tt*	*ff*	*ff*	*tt*	*ff*	*ff*	*tt*
out	*ff*	*ff*	*tt*	*tt*	*tt*	*tt*	*ff*	*ff*	*ff*	*tt*

Figure 3: Program implementing a counter modulo 2 (see example 6.2)

One problem that has to be dealt with in all attempts to define a programming paradigm based on model construction concerns the existence of models. In our setting, simple satisfiability of the given program is not enough — rather, a model has to exist for every valuation of the environment components (maybe restricted by an environment assumption). It is therefore desirable to identify necessary and sufficient conditions which ensure the success of the model construction algorithm sketched above. The representation of program clauses by DSF automata turns out very helpful in this respect. Since there are no disjunctions in node labels, it is easily verified that every possible inconsistency must be caused by at most two different program clauses containing literals p and $\neg p$ in their node labels for some $p \in \mathcal{V}_\Pi$. The desired consistency condition can therefore be obtained by constructing the product automata (which for DSFs can be defined analogously as for finite automata, taking care of the two different kinds of edges) for every pair of DSF automata representing program clauses in the given program, including product automata for two copies of every single DSF automaton, and interpreting the resulting DSF automata as temporal logic formulas by lemma 5.6. If all formulas obtained in this way follow from the environment assumptions, the algorithm will be able to construct a model. Notice that this analysis can happen "statically" and does not affect the run-time efficiency of the interpreter. A detailed study of these ideas can again be found in [14].

7 Conclusion: First-order temporal programming

In the preceding sections attention has been restricted to propositional logic, partly to obtain a technically easier presentation and partly because results such as fact 5.2 apply only to propositional νTL: there do not appear to be well-established automata-theoretic frameworks in the first-order case. However, propositional logic allows only for the development of finite-state systems of bounded state space. For practical system development, such an approach quickly becomes cumbersome if not impossible. Fortunately, most results given above readily extend to the first-order case, based either on a flexible interpretation of constants corresponding to program variables or a flexible interpretation of predicate symbols which may be viewed as modelling database relations. The underlying interpretation of the function symbols appearing in programs will always be assumed to be fixed outside of the program by referring to suitable implementations of these functions in a host programming language.

For the effectiveness of the model construction it has to be ensured that the interpreter may always deal with finite objects; in particular, all predicate extensions should be finite in every state. This requirement entails some subtle conditions on which individual variables may appear in pre-PCs. Similar conditions have been given to define "safe queries" in database theory, cf. [20]. Moreover, a "minimal-model" strategy is adopted by the interpreter algorithm which is similar to the closed-world assumption in logic programming [10]. This approach eliminates the need for negative literals appearing in pre-PC bodies, thereby reducing the size of the program clauses constituting a temporal logic program and thus helping to make them better understandable. An example program implementing a "butler" module for Dijkstra's famous dining-philosophers

(1)	$\Box(\mathrm{Hungry}(x) \rightarrow \mathrm{Waiting}(x) \ \mathsf{unless}\ \mathrm{Eat}(x) \vee \mathrm{Enough}(x))$
(2)	$\Box(\mathrm{Waiting}(x) \rightarrow (\mathrm{Left}(x)\ \mathsf{upto}\ \mathrm{Reclaim}(x \ominus 1) \vee \mathrm{Enough}(x))$
	$\qquad\qquad \mathsf{atnext}\ \neg\mathrm{Right}(x \ominus 1))$
(3)	$\Box(\mathrm{Left}(x) \wedge \neg\mathrm{Left}(x \oplus 1) \rightarrow \bigcirc(\mathrm{Right}(x) \wedge \mathrm{Eat}(x)\ \mathsf{unless}\ \mathrm{Enough}(x)))$
(4)	$\mathit{preferred} = 0\ \mathsf{upto}\ \mathrm{Reclaim}(0)$
(5)	$\Box(\mathrm{Reclaim}(x) \rightarrow \bigcirc((\mathrm{Right}(x) \wedge \mathrm{Eat}(x)\ \mathsf{unless}\ \mathrm{Enough}(x)) \wedge$
	$\qquad\qquad (\mathit{preferred} = x \oplus 1\ \mathsf{upto}\ \mathrm{Reclaim}(x \oplus 1))))$
where	$\mathrm{Reclaim}(x) \equiv \mathit{preferred} = x\ \wedge\ \mathrm{Left}(x)\ \wedge\ \mathrm{Left}(x \oplus 1)\ \wedge\ \neg\mathrm{Right}(x \oplus 1)$

Figure 4: Butler for n philosophers

problem for some fixed but unspecified number n of philosophers is shown in figure 4.

Hungry and Enough are environment predicate symbols signalling, for every philosopher x, if x is hungry or wants to finish eating. Waiting, Left and Right are internal predicates and *preferred* is a flexible constant serving as a local memory of the butler. Finally, Eat is an interface predicate owned by the butler. $\oplus$ and $\ominus$ denote addition and subtraction modulo n, respectively. All formulas are implicitly universally quantified.

Compared with more traditional programming languages, temporal logic programming offers a higher level of abstraction: State transitions do not have to be dealt with individually but they are scheduled according to the requirements expressed by the formulas constituting a program. Moreover, programs can also be seen as sets of theorems about their behaviours, allowing further assertions to be derived by temporal reasoning. The "compiler" constructing the DSF automata from the given program clauses infers the basic transitions which are then executed dynamically by the interpreter. Still, it is possible to execute such programs reasonably efficiently. (In the first-order case, some additional syntactic restrictions could be imposed to avoid combinatorial explosion of the predicate extensions.) We are currently implementing a prototype interpreter for temporal logic programs to investigate the practical validity of the approach.

Our approach is based on the assumption that the main concern of the programmer developing a reactive system should be to devise strategies for satisfying the liveness requirements of a specification which should then be implemented in a high-level but effective programming notation. This is in contrast to other proposals for temporal logic programming, in particular [3] where only syntactic restrictions are placed on admissible program clauses which do not decrease the expressiveness of the logical language. The price to be paid is a much more complicated and nondeterministic model construction algorithm which may revoke parts of the state sequence computed up to a certain stage and is therefore not suitable for interactive execution. There have also been attempts, in particular [4], to base temporal logic programming on theorem provers in a similar way as PROLOG is based on SLD-resolution. However, due to the strong incompleteness of most first-order temporal logics, severe restrictions had to be imposed on the class of admissible formulas. Also, it is not quite clear how to represent conceptually infinite runs of reactive systems by proofs which by definition are finite objects.

To obtain the forward-directed, backtracking-free model construction algorithm it has been necessary to clearly identify which syntactic and semantic restrictions to impose on admissible program clauses. Since liveness properties do not contribute to the next-state relation generated by a program, we believe they should not appear in formulas input to the interpreter. An important first step has therefore been the syntactic characterization of safety properties. Determinism has been required to avoid backtracking. Finally, the requirement on the programmer to set up a dependency order helps to make the interpreter much more efficient and tends to make programs more intellegible.

References

[1] B. Alpern, F. B. Schneider: *Recognizing safety and liveness*. Distributed Computing **2**, pp. 117–126 (1987)

[2] B. Banieqbal, H. Barringer: *Temporal Logic with Fixed Points*; in: Proc. Temporal Logic in Specification (B. Banieqbal et al., ed.). LNCS 398, pp. 62–74, Springer-Verlag (1989)

[3] H. Barringer, M. Fisher, D. Gabbay, G. Gough, R. Owens: METATEM*: A Framework for Programming in Temporal Logic*; in: J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness. LNCS 430, Springer-Verlag (1990)

[4] M. Baudinet: *Temporal Logic Programming is Complete and Expressive*; in: Proc. ACM Conf. on Princinples of Programming Languages, pp. 267–280 (1989)

[5] R. W. Floyd: *Assigning meaning to programs*; in: Proc. Symposium on Applied Mathematics **19**, pp. 19–32. American Mathematical Society (1967)

[6] C. A. R. Hoare: *An axiomatic basis for computer programming*. Communications of the ACM **12**, pp. 576–580 (1969)

[7] D. Kozen: *Results on the propositional μ-calculus*; in: E. Schmidt (ed.), Proc. 9^{th} International Colloqium on Automata, Languages, and Programming 1982, LNCS 140, pp. 348–352, Springer-Verlag. Revised Version in: Theoretical Computer Science **27**, pp. 333–354 (1983)

[8] F. Kröger: *Temporal Logic of Programs*. EATCS Monographs on Theoretical Computer Science **8**. Berlin: Springer-Verlag (1987)

[9] L. Lamport: *Proving the correctness of multiprocess programs*. IEEE Transactions on Software Engineering SE-3,2, pp. 125–143 (1977)

[10] J. W. Lloyd: *Foundations of Logic Programming*. Springer-Verlag (^{2}1987)

[11] Z. Manna, A. Pnueli: *A Hierarchy of Temporal Properties*; in: Proc. ACM Symposium on Principles of Distributed Computing (1990)

[12] Z. Manna, A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems — Specification*. New York etc.: Springer-Verlag (1992)

[13] Z. Manna, P. Wolper: *Synthesis of Communicating Processes from Temporal Logic Specifications*. ACM Transactions on Programming Languages **6**, pp. 68–93 (1984)

[14] S. Merz: *Temporal Logic as a Programming Language*. Ph.D. thesis, Ludwig-Maximilians-University Munich (1992)

[15] S. Owicki: *Axiomatic proof techniques for parallel programs*. Ph.D. thesis, Cornell University (1975)

[16] A. Pnueli: *Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends*; in: LNCS **224**, Springer-Verlag (1986)

[17] C. Stirling: *Modal and Temporal Logics*. Report ECS-LFCS-91-157, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh (1991)

[18] A. Tarski: *A lattice-theoretical fixpoint theorem and its applications*. Pacific Journal of Mathematics **5**, pp. 285–309 (1955)

[19] W. Thomas: *Automata on Infinite Objects*; in: Handbook of Theoretical Computer Science, pp. 133–191. Amsterdam: Elsevier Science Publishers B.V. (1990)

[20] J. D. Ullmann: *Principles of Database and Knowledge-Base Systems*. Computer Science Press (1988)

[21] P. Wolper: *Temporal Logic Can Be More Expressive*. Information and Control **56**, pp. 72–99 (1983)