Efficiently Executable Temporal Logic Programs

Stephan Merz

1 Introduction

Temporal logics (TL) of linear time [12, 15] offer very expressive languages that have been widely used for the specification and verification of reactive and concurrent systems. TL models describe not an unchanging world of logical truths, but an infinite sequence (of states) that naturally corresponds to the execution of a computer program over time. It is therefore a natural proposal to take a TL specification as described by a set of TL formulas, and "let it run". This requires an interpreter that accepts a set $\mathcal{S}$ of TL formulas as input, and generates a model of $\mathcal{S}$ that is then interpreted as a run of $\mathcal{S}$. Different realizations of this idea can be found, among others, in [4, 11, 19, 16, 24]. Temporal logic programming languages have also been based on the proof-as-program paradigm, either using TL theorem provers [1, 6, 20]., or constraint logic programming languages [7, 9, 10].

The designer of a TL-based programming language is mainly faced with three interrelated problems:

Complexity Even for propositional temporal logics, the satisfiability problem, while decidable, is usually PSPACE-complete [22]. It is highly undecidable (Σ_1^1-complete) for first-order temporal logics [17, 23].

System vs. environment The interpreter is not allowed to change the state of the environment. This may rule out some models of the formula, and complicates the satisfiability problem for temporal logic programs.

Interactive runs A realistic model of computation for reactive systems requires incremental model construction where the system has to respond to a finite amount of environment input, and cannot backtrack on previous reactions that have already been observed by the environment.

It is therefore necessary to find some balance between restricting the fragment of TL that is acceptable as input to the interpreter, and the efficiency of the model construction algorithm that underlies the interpreter. In the following, I will try to shed some light on the issues underlying this tradeoff. In particular, in Sections 2 and 3, I will give a syntactic characterization of propositional safety properties in the fixed-point temporal logic νTL of [3, 5], and define a restricted class of safety properties that afford a linear model-construction algorithm. The choice of νTL is convenient from a theoretical point of view because of its simple syntax, and its high expressiveness. (In fact, all conventional TL connectives [12, 15] are definable in νTL.) For a practical TL-based programming language, one may want to restrict the actual syntax to some set of predefined temporal connectives with an intuitive semantics that does not involve explicit fixpoint constructions, or at least provide a set of such connectives as predefined macros. In Section 4, I will explain how environment conditions can be generated that guarantee successful program executions, using automata-theoretic techniques. In Section 5, I extend the formalism to the first-order case. Section 6 contains a comparison with related work.

2 Deterministic safety formulas

For simplicity, I will at first restrict attention to propositional TL. It is well known that every TL formula can be written as a conjunction of a safety and a liveness property [2, 14]. By definition, if L is a liveness property, then any finite state sequence may be extended to an infinite one that satisfies L. Consequently, liveness properties cannot guide an interpreter during the construction of single (or finitely many) states to extend an already produced prefix of some run. Insisting on incremental model construction, liveness requirements have to be implemented by explicit strategies (expressed as safety formulas) that, together with basic liveness properties of the programming model, and liveness assumptions on the environment, will ensure the desired behaviour.

To illustrate this point by a simple example, consider the specification of a transmitter module that sends data received from some other module along a channel. A specification like $\Box(\mathit{DataReceived} \rightarrow \Diamond \mathit{TransmitData})$ presents the interpreter with a choice between sending the data immediately or delaying the transmission, which cannot be resolved locally. If the intended meaning of this specification is to transmit data as soon as possible, i.e. when the channel becomes free, this can—and should—be specified by a formula like $\Box(\mathit{DataReceived} \rightarrow \mathit{TransmitData}\ \mathsf{atnext}\ \mathit{ChannelFree})$, which is a safety property.

It is therefore important to syntactically identify TL formulas that express safety properties. I will base the following discussion on the temporal logic νTL [3, 5], which is defined from proposition letters like $p, q, \ldots$ and proposition variables like $X, Y, \ldots$, using the next-time operator $\bigcirc$, and the greatest fixed point operator ν. For propositional νTL, the following fact has been shown in [18]:

Fact 1 *Let F be a formula of propositional νTL. F expresses a safety property iff there is a closed propositional νTL formula G such that G is equivalent to F, and all subformulas $(\nu X\, G)$ of F occur positively.*

Another, but related concern in the definition of an executable TL is the amount of nondeterminism that is present in the specification. Since the interpreter is required to operate without backtracking, any choices left to the interpreter have to be non-essential, in the sense that none of them may lead to failure to generate a model if another choice had led to success. But such choices may as well be resolved beforehand by the programmer. The following class of *deterministic safety formulas* attempts to formalize this intuition:

Definition 2

- *The proposition constants* true *and* false *as well as the formulas p and $\neg p$, for any proposition letter p, are literals.*
- *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 is a proposition variable and F, G are pre-DSFs, then $F \wedge G$, $\bigcirc F$, $\bigcirc X$, $A \rightarrow F$ and $(\nu X\, F)$ are pre-DSFs.*
- *Every closed pre-DSF is a deterministic safety formula (DSF).*

By fact 1, every DSF defines a safety property. It is interesting to note that DSFs characterize exactly the same class of properties as the *safety automata* of [14], which are deterministic and complete Streett automata that contain a designated failure state. However, to generate models of DSFs, the following alternative automata-theoretic representation of DSFs will be more useful:

Definition 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,*
- *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 $n_0 \in N$.*

A DSF automaton $T = (N, I, S, \vartheta, \zeta, n_0)$ is called

- *normal if every edge label $\zeta(n, n')$ is a disjunction $L_1 \vee \ldots \vee L_k$ of literals,*
- *regular if $I(n) = \emptyset$ or $S(n) = \emptyset$, for every node $n \in N$,*
- *monotonic if $\vartheta(n) \subseteq \vartheta(n')$ whenever $(n, n') \in I$,*
- *complete if for every node $n \in N$ such that $I(n) \neq \emptyset$, $\bigvee_{(n,n') \in I} \zeta(n, n')$ is (propositionally) valid.*

I will represent DSF automata graphically as in Figure 5.1, where internal edges are inscribed with the edge label, whereas “crossed” arrows are used for state transitions.

The following definition states what it means for a DSF automaton to accept a temporal structure $\sigma = (s_0, s_1, \ldots)$, where a state s_i is a Boolean valuation of proposition letters:

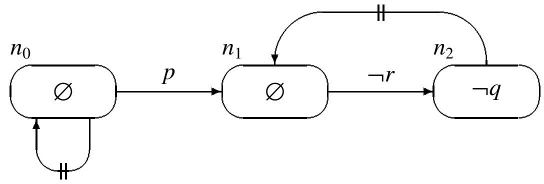


Fig. 5.1. Graphical representation of a DSF automaton

Definition 4 *The marking of a DSF automaton* $T = (N, I, S, \vartheta, \zeta, n_0)$ *for a temporal structure* $\sigma = (s_0, s_1, \dots)$ *is the* ω*-sequence* $M^\sigma = (m_0^\sigma, m_1^\sigma, \dots)$ *such that 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* $i \in \mathbb{N}$*, the labels of all nodes* $n \in m_i^\sigma$ *hold at* σ_i*:*

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

For example, the DSF automaton of Figure 5.1 accepts exactly the models of the DSF

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

(By convention, the scope of ν in temporal formulas extends as far to the right as possible.)

In fact, DSFs and DSF automata agree in expressiveness:

Theorem 5

(i) *For every DSF* φ *there is a DSF automaton* T^φ *that accepts exactly the models of* φ*, and whose size is linear in the length of* φ.

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

Proof: I only give the relevant constructions.

(i) The construction of a DSF automaton T^φ associated with a DSF φ is similar to the familiar tableau construction of temporal logic (see e.g. [26]). Formally, assume all proposition variables bound in different subformulas $(\nu X\ F)$ of φ to be pairwise different, and define the closure $cl(P)$ of any pre-clause P that appears as a subformula of φ as follows:

$cl(L) = \{L\}$ if L is a literal,

$cl(X) = \{X\} \cup cl(F_X)$ for a proposition variable X,

$cl(\bigcirc P) = \{\bigcirc P\}$, $cl(P \wedge Q) = \{P \wedge Q\} \cup cl(P) \cup cl(Q)$,

$cl(A \to P) = \{A \to P\}$, $cl(\nu X\, F_X) = \{(\nu X\, F_X)\} \cup cl(F_X)$, where F_X denotes the (unique) formula such that $(\nu X\, F_X)$ occurs as a subformula in φ. Start the construction of T^φ from an initial node n_0 labelled by $cl(\varphi)$ and inductively apply the following rules to construct new nodes and edges. Whenever a node with the required label already exists, only a new edge has to be inserted; if the two nodes are already connected by an edge, the new label will be added as a disjunct to the existing edge label:

(i) For every node n whose label contains a formula $A \rightarrow P$, construct a node n' labelled by $cl(P)$ and an internal edge $(n, n') \in I$ labelled by A.

(ii) For every node n whose label ϑ contains a formula $\bigcirc P$ construct a node n' labelled by

$$\bigcup\{cl(P) \mid \bigcirc P \in \vartheta\}$$

and a state transition $(n, n') \in S$.

T^φ is then obtained from the resulting graph by restricting all node labels to literals.

(ii) To construct φ^T for a DSF automaton $T = (N, I, S, \vartheta, \zeta, n_0)$ whose set of nodes is $N = \{n_0, \ldots, n_k\}$, let $X_0, \ldots, X_k$ be proposition variables, and define the formulas ψ_j^X and δ_j^X for $0 \le j \le k$ and every subset $X \subseteq \{X_0, \ldots, X_k\}$ of the proposition variables by

$$\psi_j^X \equiv \vartheta(n_j) \;\wedge \bigwedge_{(n_j,n_l)\in I} (\zeta(n_j, n_l) \rightarrow \delta_l^X) \;\wedge \bigwedge_{(n_j,n_l)\in S} \bigcirc\delta_l^X,$$

$$\delta_j^X \equiv \left\{ \begin{array}{ll} X_j & \text{if } X_j \in X \\ (\nu X_j\ \psi_j^{X\,\cup\,\{X_j\}}) & \text{otherwise} \end{array} \right\}.$$

Now, define $\varphi^T :\equiv \delta_0^\emptyset \equiv \nu X_0\ \psi_0^{\{X_0\}}$. Induction on $\{X_0, \ldots, X_k\} \setminus X$ shows that all formulas ψ_i^X and δ_i^X are well-defined pre-DSFs whose free proposition variables are among the variables in X. In particular, φ^T is a DSF. ◊

By standard automata-theoretic constructions, it is easy to see that for every DSF automaton T there is a regular, monotonic, normal and complete DSF automaton T' that accepts exactly the same temporal structures as T, and whose size is linear in the size of T (the construction of a normal DSF automaton T' is linear only if all edge labels of T are in conjunctive normal form, but this is not practically relevant since edge labels will usually be simple).

3 Program Clauses

DSF automata provide an operational representation of DSFs if we interpret edge and node labels as tests to be performed, and assignments of (boolean) values to proposition letters, respectively. A reactive system, however, may change the values of system-owned state components only, whereas environment-owned components may only be read. Formally, distinguish two sets $\mathcal{V}_E$ and $\mathcal{V}_\Pi$ of proposition letters that model, respectively, environment and system owned state components, and restrict proposition letters $p \in \mathcal{V}_E$ to appear only in antecedents of program clauses. Proposition letters from $\mathcal{V}_\Pi$ may appear both in antecedents and pre-clause bodies. However, to rule out circular definitions where the values of two system owned state components may mutually depend on one another, the programmer will be required to provide a partial order $<$ on the set $\mathcal{V}_\Pi$. This *dependency order* is then extended to arbitrary literals $L_1 \equiv (\neg)p$, $L_2 \equiv (\neg)q$ by

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

The following definition defines *program clauses* as the subset of DSFs such that $p < q$ holds whenever the value that will be assigned to q in some state may depend on the value of p in that same state. This additional condition simplifies the interpreter algorithm. In the first-order case, it will be essential to ensure that the construction of each single state in the model construction algorithm will actually terminate.

Definition 6

- *Every literal L is an antecedent. $read(L) = \{p\}$ if $L \equiv (\neg)p$ for some $p \in \mathcal{V}_E \cup \mathcal{V}_\Pi$, $read(L) = \varnothing$ for $L \in \{\mathsf{true}, \mathsf{false}\}$.*
- *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 and X is a proposition variable then $P \wedge Q$, $\bigcirc P$, $\bigcirc X$ and $(\nu X\, P)$ are pre-PCs where $write(P \wedge Q) = write(P) \cup write(Q)$, $write(\bigcirc P) = write(\bigcirc X) = \varnothing$ and $write(\nu X\, P) = write(P)$.*
- *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 $A \to P$ is a pre-PC. $write(A \to P) = write(P)$.*
- *Every closed pre-PC (that does not contain free proposition variables) is a program clause (PC).*

The interpreter algorithm of Figure 5.2 attempts to construct a model for a set Φ of program clauses by incremental calculation of the markings of normal DSF automata associated with the clauses in Φ, starting from the environment-defined values of proposition letters $p \in \mathcal{V}_E$, and defining the valuation of $q \in \mathcal{V}_\Pi$ according to the node labels. Since no backtracking occurs, the interpreter is suitable for the interactive execution of reactive programs. It is easy to see that each node of every DSF automaton is visited at most once, and each edge is tested at most once per iteration of the algorithm. Therefore, both the space and time complexity of the construction of any single state are linear in the size of the normal DSF automata representing the program Φ. By theorem 5, that size is itself linear in the length of the formulas in Φ, provided that all antecedents are in conjunctive normal form.

Example 7 Figure 5.3 illustrates the operation of the algorithm by example of a program that monitors an environment-owned boolean variable *in*, counts (modulo 2) the number of times *in* has changed from *ff* to *tt*, and displays this count in the system-owned boolean variable *out* (where 0 and 1 represent *ff* and *tt*, respectively). *lastin* is an auxiliary (internal) boolean variable containing the value of *in* at the previous state.

Let $\Phi = \{\varphi^1, \ldots, \varphi^q\}$ be a set of temporal program clauses, and $\varepsilon = (\varepsilon_0, \varepsilon_1, \ldots)$ be a temporal structure assigning values $\varepsilon_i(p) \in \{tt, ff\}$ to the environment state components $p \in \mathcal{V}_E$. Let $T^k = (N^k, I^k, S^k, \vartheta^k, \zeta^k, n_0^k)$ be a normal DSF automaton associated with the program clause φ^k. Construct the temporal structure $\sigma(\Phi, \varepsilon) = (\sigma_0, \sigma_1, \ldots)$ as follows:

A *configuration* of $T^1, \ldots, T^q$ consists of a q-tuple $\kappa = (\nu^1, \ldots, \nu^q)$ of sets $\nu^k \subseteq N^k$ of nodes of the DSF automata, and a set $\mathcal{E}$ of internal edges that have yet to be inspected. Let "$L \in \zeta^k(n, n')$" denote that the literal L appears as a disjunct in the edge label $\zeta^k(n, n')$. The algorithm defines a sequence of configurations $(\kappa_i^0, \kappa_i^1, \ldots)$ for every state σ_i of the temporal structure $\sigma(\Phi, \varepsilon)$ under construction:

Initialization: Let $i = 0, j = 0$, and define the initial configuration $\kappa_0^0 := (\{n_0^1\}, \ldots, \{n_0^q\})$.
State construction: For the initial configuration $\kappa_i^0 = (\nu^1, \ldots, \nu^q)$, let

$$\mathcal{E} := \bigcup_{k=1}^{q} \{(n, n', L) \mid n \in \nu^k, (n, n') \in I^k, L \in \zeta^k(n, n')\}$$

be the initial set of edges to explore. Define

$$\sigma_i(p) := \left\{ \begin{array}{ll} \varepsilon_i(p) & \text{if } p \in \mathcal{P}_E \text{ is an environment component} \\ tt & \text{if } p \in \vartheta^k(n) \text{ for some } n \in \nu^k \quad (1 \le k \le q) \\ ff & \text{if } \neg p \in \vartheta^k(n) \text{ for some } n \in \nu^k \quad (1 \le k \le q) \\ \perp & \text{else} \end{array} \right\}.$$

Complete the construction of σ_i as follows:

1. While $\mathcal{E} \neq \emptyset$, iterate the following steps:
   (a) Set $j := j + 1$. Choose some $(n_k, n_k', L) \in \mathcal{E}$ (where $n_k, n_k' \in N^k$) such that for no $(n_l, n_l', L') \in \mathcal{E}$ it holds that $L' < L$ w.r.t. the dependency order $<$ on literals. If $L \equiv (\neg)p$ and $\sigma_i(p) = \perp$, then arbitrarily define $\sigma_i(p) := tt$ or $\sigma_i(p) := ff$.
   (b) If $L \equiv p$ and $\sigma_i(p) = tt$, or if $L \equiv \neg p$ and $\sigma_i(p) = ff$, or if $L \equiv \mathsf{true}$, then $\kappa_i^j := (\nu^1, \ldots, \bar{\nu}^k, \ldots, \nu^q)$ where $\bar{\nu}^k = \nu^k \cup \{n_k'\}$. In this case, define $\sigma_i(p) := tt$ for every $p \in \vartheta^k(n_k')$, and $\sigma_i(p) := ff$ for every p such that $\neg p \in \vartheta^k(n_k')$. Set

$$\begin{aligned} \mathcal{E} := {} & (\mathcal{E} \setminus \{(n_k'', n_k', L')\}) \\ & \cup \{(n_k', n_k'', L') \mid (n_k', n_k'') \in I^k, n_k'' \notin \bar{\nu}^k, L' \in \zeta^k(n_k', n_k'')\} \end{aligned}$$

   Else, $\kappa_i^j := \kappa_i^{j-1}$, and $\mathcal{E} := \mathcal{E} \setminus \{(n_k, n_k', L)\}$.
2. For all $p \in \mathcal{V}_\Pi$ such that $\sigma_i(p) = \perp$, arbitrarily define $\sigma_i(p) := tt$ or $\sigma_i(p) := ff$.
3. Let $\kappa_i^j = (\nu^1, \ldots, \nu^q)$ be the current configuration. Set $\kappa_{i+1}^0 := (\bar{\nu}^1, \ldots, \bar{\nu}^q)$ where $\bar{\nu}^k = \bigcup_{n \in \nu^k} S(n)$. Set $i := i + 1$ and $j := 0$, and repeat the state construction.

Fig. 5.2. Interpreter algorithm for propositional program clauses

The definition of the dependency order is irrelevant for this example. DSF automata representing the program clauses as well as the first ten states of an example run are shown as well. The program uses the temporal operators

$$\Box F \equiv \nu X.F \wedge \bigcirc X$$

$$F \;\mathsf{upto}\; G \equiv \nu X\, F \;\wedge\; (\neg G \to \bigcirc X)$$

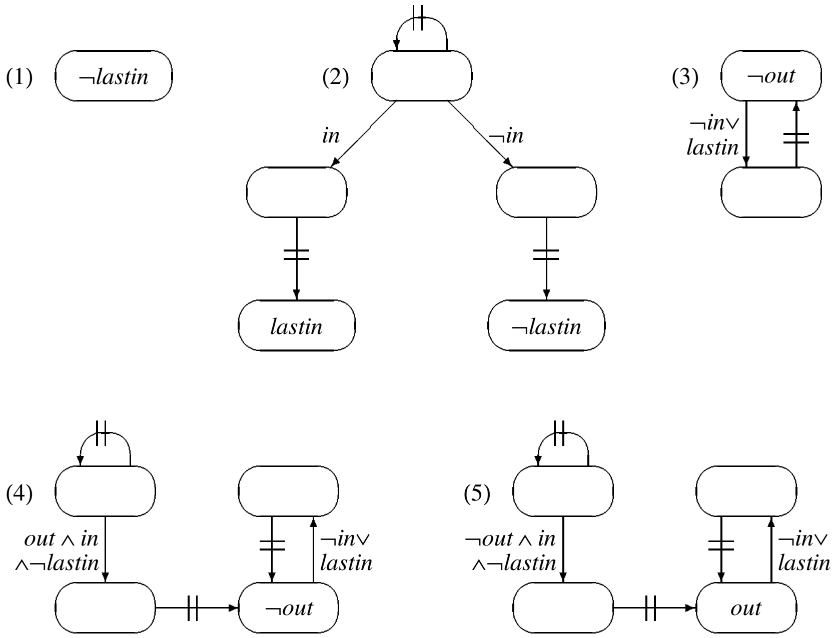
(1) $\neg lastin$

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

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

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

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



	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*

Fig. 5.3. Counter modulo 2 (see example 7)

4 Conflict automata

Obviously, the interpreter can construct a model for a given TL program Φ only if Φ is satisfiable—a nontrivial problem, as shown in [22, 26]. In fact, a stronger condition is required, since the valuation of the environment state components is left unchanged by the interpreter: For a given environment valuation ε, there has to be a model of Φ that agrees with ε on $\mathcal{V}_E$. Fortunately, we can use the automata-theoretic representation of Φ by DSF automata to generate necessary and sufficient conditions (i.e., environment assumptions) for the model construction to succeed. The essential definition is that of a product automaton for DSF automata.

Definition 8 *For regular, normal DSF automata $T^1 = (N^1, I^1, S^1, \vartheta^1, \zeta^1, n_0^1)$ and $T^2 = (N^2, I^2, S^2, \vartheta^2, \zeta^2, n_0^2)$, the conflict automaton $T = (N, I, S, \vartheta, \zeta, n_0)$ is defined as follows:*

$N \subseteq N^1 \times N^2$ is the smallest set of nodes satisfying the following conditions:

- $n_0 := (n_0^1, n_0^2) \in N$
- If $(n^1, n^2) \in N$ and $I^1(n^1) \neq \emptyset$, then $I^1(n^1) \times \{n^2\} \subseteq N$, $I(n^1, n^2) = I^1(n^1) \times \{n^2\}$, $S(n^1, n^2) = \emptyset$, and $\zeta((n^1, n^2), (n, n^2)) = \zeta^1(n^1, n)$ for every $(n, n^2) \in I(n^1, n^2)$.
- If $(n^1, n^2) \in N$ and $I^1(n^1) = \emptyset$, but $I^2(n^2) \neq \emptyset$, then $\{n^1\} \times I^2(n^2) \subseteq N$, $I(n^1, n^2) = \{n^1\} \times I^2(n^2)$, $S(n^1, n^2) = \emptyset$, and $\zeta((n^1, n^2), (n^1, n)) = \zeta^2(n^2, n)$ for every $(n^1, n) \in I(n^1, n^2)$.
- If $(n^1, n^2) \in N$ and $I^1(n^1) = I^2(n^2) = \emptyset$, then $S^1(n^1) \times S^2(n^2) \subseteq N$, $I(n^1, n^2) = \emptyset$ and $S(n^1, n^2) = S^1(n^1) \times S^2(n^2)$.

The node labelling ϑ of T is computed as follows: Let all nodes $(n^1, n^2) \in N$ be unmarked, and let initially $\vartheta(n^1, n^2) := \emptyset$. While there exists some unmarked node $(n^1, n^2) \in N$ such that $\vartheta^1(n^1) \cup \vartheta^2(n^2) \cup \vartheta(n^1, n^2)$ contains either false or a complementary pair $p, \neg p$ of literals:

- Set $\vartheta(n^1, n^2) := \vartheta(n^1, n^2) \cup \{\mathsf{false}\}$.
- For all $((n', n''), (n^1, n^2)) \in I$ where $\zeta((n', n''), (n^1, n^2)) = L_1 \vee \ldots \vee L_k$, set $\vartheta(n', n'') := \vartheta(n', n'') \cup \{\sim L_1, \ldots, \sim L_k\}$ where $\sim L_i$ is $\neg p$ if $L_i \equiv p$, $\sim L_i$ is p if $L_i \equiv \neg p$, $\sim\mathsf{true} = \mathsf{false}$, and $\sim\mathsf{false} = \mathsf{true}$.
- For all $((n', n''), (n^1, n^2)) \in S$, set $\vartheta(n', n'') := \vartheta(n', n'') \cup \{\mathsf{false}\}$.
- Mark the node (n^1, n^2) as processed.

Nodes marked in the computation of ϑ as well as those all of whose $(I \cup S)^*$-successors have an empty node label, except the initial node, may be deleted from T. $\Diamond$

Example 9 Figure 5.4 illustrates the construction of definition 8 for two regular, normal and monotonic DSF automata corresponding to the DSFs

$$\begin{aligned}
\varphi_1 &\equiv r_1 \rightarrow (p \wedge q) \;\mathsf{unless}\; r_2 \\
&\equiv r_1 \rightarrow \nu X\, \neg r_2 \rightarrow (p \wedge q \wedge \bigcirc X)
\end{aligned}$$

$$\begin{aligned}
\varphi_2 &\equiv (\neg q \;\mathsf{atnext}\; p) \;\mathsf{atnext}\; r_3 \\
&\equiv \nu Y\, (\neg r_3 \rightarrow \bigcirc Y) \;\wedge\; (r_3 \rightarrow \nu X\, (p \rightarrow \neg q) \wedge (\neg p \rightarrow \bigcirc X))
\end{aligned}$$

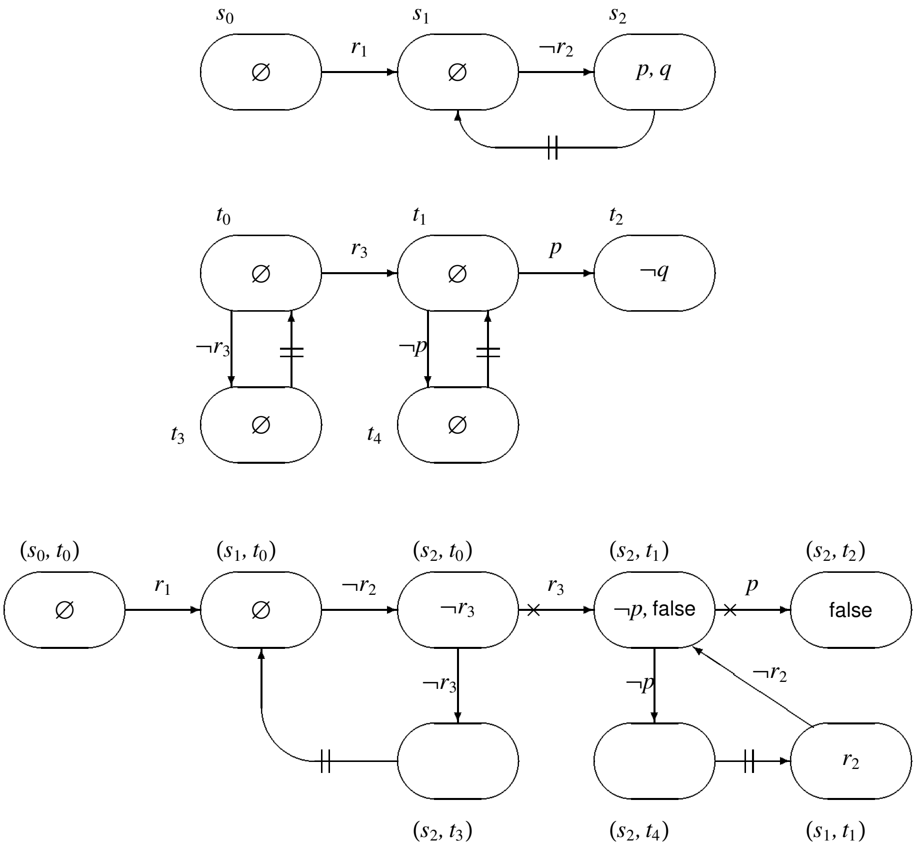


Fig. 5.4. Example for the construction of a conflict graph

Notice how the inconsistency detected at node (s_2, t_2) is propagated back, causing the label of (s_2, t_1) to become inconsistent. After simplification, the DSF associated with the conflict automaton by lemma 5 becomes

$$r_1 \to \nu X\,(\neg r_2 \to \neg r_3 \wedge \bigcirc X) \;\equiv\; r_1 \to \neg r_3 \text{ unless } r_2.$$

The two following lemmas express that conflict automata characterize the conditions for successful model construction:

Lemma 10 *Let T^1 and T^2 be two regular and normal DSF automata, and let T be their conflict automaton. Every temporal structure accepted by both T^1 and T^2 is also accepted by T.*

Proof: Let σ be a temporal structure, and denote by $M^1 = (m^1_0, m^1_1, \ldots)$, $M^2 = (m^2_0, m^2_1, \ldots)$, and $M = (m_0, m_1, \ldots)$, the markings over σ of T^1, T^2, and T, respectively. It is easy to prove that, for all nodes $n^1 \in N^1$, $n^2 \in N^2$ and every $i \geq 0$, it holds that $(n^1, n^2) \in m_i$ implies $n^1 \in m^1_i$ and $n^2 \in m^2_i$.

Using this auxiliary assertion and the assumption that σ is accepted by both T^1 and T^2, one proves by induction on the definition of the node labelling ϑ of T that for all nodes $(n^1, n^2) \in N$ and every $i \geq 0$ it holds that $(n^1, n^2) \in m_i$ implies $\sigma_i \models \vartheta(n^1, n^2)$. $\Diamond$

Lemma 11 *Let T^1 and T^2 be two regular, normal, complete and monotonic DSF automata, and let T be the conflict automaton of T^1 and T^2. Let σ be some temporal structure that is accepted by T, and denote by $M^1 = (m^1_0, m^1_1, \ldots)$ and $M^2 = (m^2_0, m^2_1, \ldots)$ the markings of T^1 and T^2, respectively, for σ. Then for no $i \geq 0$ there are nodes $n^1 \in m^1_i$, $n^2 \in m^2_i$ such that $\vartheta^1(n^1) \cup \vartheta^2(n^2)$ contains a complementary pair $p, \neg p$ of literals.*

Proof: For a complete DSF automaton T, a node $n_1 \in N$, and a state σ_i of a temporal structure, call $\pi = (n_1, n_2, \ldots, n_k)$ a fullpath for n_1 and σ_i in T if π is a maximal I^*-path in T starting in n_1 such that all edge labels along the path are satisfied in σ_i. Since T is complete, it follows that $I(n_k) = \emptyset$. Let T^1 and T^2 be regular, normal, and complete DSF automata, T their conflict automaton, and σ any temporal structure. It is easy to prove that for every $i \geq 0$, whenever $\pi^1 = (n^1_1, \ldots, n^1_k)$ and $\pi^2 = (n^2_1, \ldots, n^2_k)$ are two fullpaths for n^1_1 and n^2_1 and σ_i in T^1 and T^2, respectively, such that $n^1_1 \in m^1_i$ and $n^2_1 \in m^2_i$, then for some u, v we must have $(n^1_u, n^2_v) \in m_i$ (where $M = (m_0, m_1, \ldots)$ denotes the marking of T over σ). Since T is accepted by σ, it follows that $\mathsf{false} \notin \vartheta(n^1_u, n^1_v)$, hence $\vartheta^1(n^1_u) \cup \vartheta^2(n^2_v)$ cannot contain a complementary pair of literals. Since T^1 and T^2 were also assumed to be monotonic, the same follows for $\vartheta^1(n^1) \cup \vartheta^2(n^2)$. $\Diamond$

Given two regular, normal, complete, and monotonic DSF automata T^1 and T^2 for program clauses φ^1 and φ^2, the two preceding lemmas imply that the conflict automaton of T^1 and T^2 characterizes a necessary and sufficient condition for the model construction for φ^1 and φ^2 to succeed. In the more general case of a set Φ of program clauses, it is easy to see that conflict automata for every pair of program clauses of Φ, including two distinct copies for every single program clause, precisely identify the conditions for runs of Φ to succeed. This fact is due to the absence of disjunctions in pre-clause bodies (and hence in node labels), i.e. to the high degree of determinism required of program clauses. These ideas are formally captured by the following definition of *compatibility* that allows for environment assumptions, as is appropriate for a rely-guarantee style of program development.

Definition 12 *Let $\Phi = \{\varphi_1, \ldots, \varphi_q\}$ be a set of program clauses, and Ψ be some closed νTL formula over $\mathcal{V}_E$. Denote by $T_1, \ldots, T_q$ regular, normal, monotonic, and complete DSF automata corresponding to $\varphi_1, \ldots, \varphi_q$. Φ is called Ψ-compatible if for all $1 \leq k \leq q$ and $1 \leq j \leq k$, the DSFs corresponding to the conflict automata of T_j and T_k follow from Ψ and $\{\varphi_1, \ldots, \varphi_{k-1}\}$.*

By an induction on the number of steps taken by the interpreter algorithm, the following theorem has been shown in [18]:

Theorem 13 *Let Φ and Ψ be as in definition 12.*

(i) *Φ is Ψ-compatible iff for every model $\varepsilon = (\varepsilon_0, \varepsilon_1, \ldots)$ of Ψ there is a model of Φ that agrees with ε on $\mathcal{V}_E$.*
(ii) *If Φ is Ψ-compatible and ε is a model of Ψ, then the interpreter algorithm of Figure 5.2 constructs a model σ of Φ that agrees with ε on $\mathcal{V}_E$.*

Notice that the property of being Ψ-compatible w.r.t. some given environment assumption Ψ can be decided "statically" by an analysis of the given program Φ. For the example program of Figure 5.3, all conflict automata turn out to be trivial, hence the program is Ψ-compatible w.r.t. any environment condition Ψ.

5 First-order program clauses

Propositional TL can describe systems of bounded state space. When it is impossible or inconvenient to bound the state space a priori, first-order TL offers the appropriate expressiveness. Fortunately, most notions introduced above generalize readily to the first-order case. First-order TL can represent local state using either *flexible variables*, which correspond to program variables of conventional imperative programming languages, or *flexible predicates*, which are appropriate to model sets of data such as database relations. On the other hand, I assume all function (and constant) symbols, as well as equality to be rigid.

The universe and the interpretation of the function symbols are provided by an underlying first-order model. In terms of an actual interpreter realization, this means that they will be implemented in a suitable host programming language. Every state of a temporal structure interprets the predicate symbols and the flexible variables of the language. First-order νTL also contains a set of *rigid variables* [15] to bind data values, as in classical first-order logic. I will allow quantification over rigid variables only. For a term t, let $FRV(t)$ and $FFV(t)$ denote, respectively, the sets of rigid and flexible variables occurring in t, and similarly for a formula F.

The overall structure of first-order DSFs and program clauses is the same as in the propositional case. As is conventional in logical and rule-based programming languages, all rigid variables that occur in clauses are interpreted as being in the scope of an implicit universal quantifier. This ensures that all program clauses express safety properties, using the following fact [18] that extends Fact 1:

Fact 14 *Let F be a first-order νTL formula such that no subformula (νXG) of F appears either negatively or in the scope of a quantifier of existential force. Then F defines a safety property.*

For first-order TL programs that contain flexible predicate symbols, it has to be ensured that the predicate extensions at each state remain finite; otherwise the model could not be effectively presented. This requirement entails some conditions on the rigid variables appearing in program clauses, similar to the definition of safe queries in database theory [25]. Since the universe will in general be (conceptually) infinite, it follows that the complements of predicate extensions will not be explicitly representable. Hence, the interpreter will adopt a "minimal-model" approach during the state construction: For an n-ary predicate P and arbitrary elements $d_1, \dots, d_n$ of the universe, $(d_1, \dots, d_n) \in \sigma_i(P)$, for any state σ_i, will hold only if this follows from an instance of some program clause. This convention eliminates the need for negative literals to appear in pre-clauses. Negative literals may, however, appear in antecedents, where they are interpreted as tests.

Formally, first-order DSFs are defined as follows:

Definition 15

- *For any term t, define the set $Bind(t)$ by $Bind(t) = \{y\}$ if $t \equiv y$ is a rigid variable, and $Bind(t) = \emptyset$ else. Intuitively, $Bind(t)$ denotes the set of rigid subject variables for which t may establish a binding if it appears in an antecedent.*
- *Atomic formulas $P(t_1, \dots, t_m)$ or $t = t'$ as well as their negations and the propositional constants* true *and* false *are literals. For a literal L, the set $Bind(L)$ is defined by*

$$Bind(P(t_1, \dots, t_m)) = (\bigcup_{i=1}^{m} Bind(t_i)) \setminus (\bigcup_{Bind(t_j)=\emptyset} FRV(t_j)),$$
$$Bind(t = t') = Bind(t) \setminus FRV(t'),$$
$$Bind(\neg P(t_1, \dots, t_m)) = \emptyset, \qquad Bind(t \neq t') = \emptyset,$$
$$Bind(\mathsf{true}) = \emptyset, \qquad Bind(\mathsf{false}) = \emptyset.$$

- *Every literal is an antecedent. If A, B are antecedents then $A \wedge B$ and $A \vee B$ are also antecedents where $Bind(A \wedge B) = Bind(A \vee B) = Bind(A) \cap Bind(B)$.*
- *Atomic formulas of the form $P(t_1, \dots, t_m)$ or $t = t'$ are pre-DSFs. Define the set $Open(P(t_1, \dots, t_m)) = \bigcup_{i=1}^{m} FRV(t_i)$, and $Open(t = t') = FRV(t) \cup FRV(t')$. Intuitively, $Open(P)$ denotes the set of rigid variables that have to be bound before the pre-DSF P is evaluated.*
- *If A is an antecedent, X is a proposition variable and F, G are pre-DSFs then $F \wedge G$, $\bigcirc F$, $\bigcirc X$, $A \rightarrow F$ and $(\nu X\ F)$ are pre-DSFs where $Open(F \wedge G) = Open(F) \cup Open(G)$, $Open(\bigcirc F) = Open(\nu X\ F) = Open(F)$, $Open(\bigcirc X) = \emptyset$ and $Open(A \rightarrow F) = (FRV(A) \cup Open(F)) \setminus Bind(A)$.*
- *If F is a pre-DSF that does not contain any free proposition variable and where $Open(F) = \emptyset$, then F is a deterministic safety formula (DSF).*

First-order DSFs may be represented in an automata-theoretic framework similar to the propositional DSF automata introduced above. The only extension is to record the set of rigid variables bound at each node and to impose variable conditions on literals that appear in node and edge labels, corresponding to those for DSFs. First-order program clauses are then again defined by distinguishing between state components owned by the environment and those owned by the system. In particular, let $\mathcal{V}_E$, $\mathcal{V}_\Pi$, $\mathcal{P}_E$ and $\mathcal{P}_\Pi$, respectively, denote the sets of flexible variables and predicate symbols owned by environment and system. Again, the programmer has to provide a partial dependency order $<$ on the set $\mathcal{V}_\Pi \cup \mathcal{P}_\Pi$ of system-owned state components. Atomic pre-clauses may be either of the form $P(t_1, \dots, t_m)$ for $P \in \mathcal{P}_\Pi$ or of the form $v = t$ for $v \in \mathcal{V}_\Pi$ (resembling an assignment to the program variable v) provided that $v' < P$ (resp., $v' < v$) holds for every flexible variable v' occurring in any term t_i (resp., in t). The definition of PCs then proceeds inductively as in the propositional case.

The definition of conflict automata to generate formulas that guarantee the success of the interpreter algorithm is in fact easier than in the propositional case: Since negative literals have been ruled out in pre-clause bodies, the only possible inconsistencies may arise from assignments of different values to the same flexible variable. If two nodes n^1 and n^2 of the original DSF automata contain such assignments $v = t$ and $v = t'$, the corresponding node (n^1, n^2) of the product automaton will contain $t = t'$ in its node label.

Having defined conflict automata, the condition of Ψ-compatibility of some program Φ w.r.t. an environment assumption Ψ remains as before. Given a program Φ, all models generated by the interpreter will have minimal extensions of flexible system predicates, and therefore satisfy additional properties over the logical semantics of Φ. To make the logical and operational semantics of Φ agree, a notion of *program completion* is defined, similarly as for negation-free classical logic programs [13], using the automata-theoretic representation of program clauses. The compatibility criterion can then be relativized to the completed program.

The interpreter algorithm is a relatively straightforward extension of the propositional interpreter of Figure 5.2. For first-order programs without flexible predicates, the complexity to build each single state of the structure is still linear in the size of the automata representing the program clauses. However, if flexible predicate symbols are allowed, it is easy to find examples that exhibit exponential growth of the predicate extensions along the state sequence. It may be worthwhile to impose further syntactic conditions to prevent such undesired behaviour.

$$
\begin{array}{ll}
(1) & \Box(\mathit{Hungry}(x) \rightarrow \mathit{Waiting}(x) \;\mathsf{unless}\; \mathit{Eat}(x) \vee \mathit{Enough}(x)) \\
(2) & \Box(\mathit{Waiting}(x) \rightarrow (\mathit{Left}(x) \;\mathsf{upto}\; \mathit{Reclaim}(x \ominus 1) \vee \mathit{Enough}(x)) \\
 & \qquad\qquad\qquad \mathsf{atnext}\; \neg \mathit{Right}(x \ominus 1)) \\
(3) & \Box(\mathit{Left}(x) \wedge \neg \mathit{Left}(x \oplus 1) \rightarrow \bigcirc(\mathit{Right}(x) \wedge \mathit{Eat}(x) \;\mathsf{unless}\; \mathit{Enough}(x))) \\
(4) & \mathit{preferred} = 0 \;\mathsf{upto}\; \mathit{Reclaim}(0) \\
(5) & \Box(\mathit{Reclaim}(x) \rightarrow \bigcirc((\mathit{Right}(x) \wedge \mathit{Eat}(x) \;\mathsf{unless}\; \mathit{Enough}(x)) \wedge \\
 & \qquad\qquad\qquad (\mathit{preferred} = x \oplus 1 \;\mathsf{upto}\; \mathit{Reclaim}(x \oplus 1)))) \\
\multicolumn{2}{l}{\text{where } \mathit{Reclaim}(x) \equiv \mathit{preferred} = x \;\wedge\; \mathit{Left}(x) \;\wedge\; \mathit{Left}(x \oplus 1) \;\wedge\; \neg \mathit{Right}(x \oplus 1),} \\
 & x \oplus 1 \equiv (x+1) \bmod n, \quad x \ominus 1 \equiv (x-1) \bmod n
\end{array}
$$

Fig. 5.5. Butler for n philosophers

As an example for a first-order temporal logic program, figure 5.5 shows an implementation of a "butler" module for the dining-philosopher program for an arbitrary number n of philosophers. Here, *Hungry* and *Enough* are environment-owned unary predicate symbols, *Waiting*, *Left*, *Right*, and *Eat* are system-owned unary predicate symbols, and *preferred* is a system-owned flexible variable. The dependency order is defined by $\mathit{Eat} < \mathit{Waiting} < \mathit{Right} < \mathit{Left}$.

6 Discussion

Compared to conventional programming languages for reactive systems, temporal logic programming promises a higher level of abstraction that relieves the developer from explicitly stating the next-state relation. In our approach, it is inferred by the compiler that constructs the DSF automata. Another important feature is the absence of backtracking, because this allows for incremental model construction. As a sanity check on programs, it is possible to automatically generate formulas that guarantee successful runs. The developer should then prove these formulas from given environment assumptions. (In the propositional case, this proof can be automated, since propositional νTL is decidable).

By comparison, the languages proposed in [19] or [24] are basically similar to conventional programming languages, and offer only few higher-level constructs. At the other end of the spectrum, the language METATEM [4] offers the expressiveness of full linear-time temporal logic, and imposes only a syntactic normal form on admissible formulas. Consequently, the interpreter algorithm becomes much more complex, since it needs to detect failure of the model construction, and backtrack if it does. In particular, the environment and the system cannot truly run interactively, because the environment cannot be sure that the system's reactions will remain stable. In general, early detection of failure situations requires theorem-proving abilities and becomes an undecidable problem. Another tradeoff that has been resolved differently concerns the presence of past-time connectives in antecedents. Since these are allowed in METATEM, the interpreter has to store the entire computation history. In contrast, our interpreter can evaluate all antecedents solely on the basis of the present state. On the other hand, the programmer may have to define auxiliary state components that serve as a memory (such as the predicate *Waiting* in the dining philosophers example).

The languages TEMPLOG [1, 6] and CHRONOLOG [20] aim to base a temporal logic programming language on theorem proving in a similar way as PROLOG is based on SLD-resolution. In such approaches, given the strong incompleteness of first-order temporal logics, the class of admissible formulas has to be restricted severely. Also, it is not so clear intuitively how to generate infinite runs of reactive systems from finite proofs than it is to interpret temporal structures as such runs. It has recently become clear [7, 9, 10] that the powerful mechanisms of constraint logic programming languages can be employed to perform temporal logic proofs, thereby obviating the need for special-purpose TL theorem provers.

Acknowledgements

It is a pleasure to express my gratitude toward all those who by constructive criticism have helped to form the ideas or to improve the presentation. In particular, I want to thank Jean Paul Bahsoun, Luis Fariñas del Cerro, Hans Gaßner, Fred Kröger, Frank Leßke, Holger Schlingloff, Corinne Servières, and Martin Wirsing, as well as the other members of the nonclassical logics groups at Munich and Toulouse.

References

1. M. Abadi: *Temporal-Logic Theorem Proving*. Ph.D. thesis, Department of Computer Science, Stanford University (1987)
2. B. Alpern, F. B. Schneider: *Recognizing safety and liveness*. Distributed Computing **2**, pp. 117–126 (1987)
3. 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)
4. 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). See also [8].
5. H. Barringer, R. Kuiper, A. Pnueli: *Now You May Compose Temporal Logic Specifications*. Proc. Sixteenth ACM Symposium on the Theory of Computing, pp. 51–63 (1984)
6. M. Baudinet: *Temporal Logic Programming is Complete and Expressive*; in: Proc. ACM Conf. on Princinples of Programming Languages, pp. 267–280 (1989)
7. C. Brzoska: *Temporal Logic Programming with Metric and Past Operators*. Proc. Workshop on Executable Modal and Temporal Logics, in this volume.
8. M. Fisher: *Concurrent* METATEM *— The Language and its Applications*. Proc. Workshop on Executable Modal and Temporal Logics, IJCAI 1993, in this volume.
9. T. Fruehwirth: *Temporal Logic and Annotated Constraint Logic Programming*. Proc. Workshop on Executable Modal and Temporal Logics, IJCAI 1993, in this volume.
10. J. Koehler, R. Treinen: *Constraint Deduction in an Interval-based Temporal Logic*. Proc. Workshop on Executable Modal and Temporal Logics, IJCAI 1993, in this volume.
11. S. Kono: *A Combination of Clausal and Non Clausal Temporal Logic Program*. Proc. Workshop on Executable Modal and Temporal Logics, IJCAI 1993, in this volume.
12. F. Kröger: *Temporal Logic of Programs*. EATCS Monographs on Theoretical Computer Science **8**. Berlin: Springer-Verlag (1987)
13. J. W. Lloyd: *Foundations of Logic Programming*. Springer-Verlag (^{2}1987)
14. Z. Manna, A. Pnueli: *A Hierarchy of Temporal Properties*; in: Proc. ACM Symposium on Principles of Distributed Computing (1990)
15. Z. Manna, A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems — Specification*. New York etc.: Springer-Verlag (1992)
16. Z. Manna, P. Wolper: *Synthesis of Communicating Processes from Temporal Logic Specifications*. ACM Transactions on Programming Languages **6**, pp. 68–93 (1984)
17. S. Merz: *Decidability and Incompleteness Results for First-Order Temporal Logics of Linear Time*. Journal of Applied Non-Classical Logics **2**(2) (1992)
18. S. Merz: *Temporal Logic as a Programming Language*. Ph.D. thesis, Ludwig-Maximilians-Universität (1992)

19. B. Moszkowski: *Executing Temporal Logic Programs*. Cambridge (UK): Cambridge University Press (1986)
20. M. A. Orgun, W. W. Wadge: *Towards a unified theory of intensional logic programming*. Journal of Logic Programming, **13**(4), pp. 413ff (1992)
21. A. Pnueli: System Specification and Refinement in Temporal Logic. Proc. Conf. on Foundations of Software Technology and Theoretical Computer Science. Springer LNCS 652, pp. 1–38 (1992)
22. A. P. Sistla, E. M. Clarke: *The Complexity of Propositional Linear Temporal Logic*. Journal of the ACM **32**(3), pp. 733–749 (1985)
23. A. Szałas: *A Complete Axiomatic Characterization of First-Order Temporal Logic of Linear Time*. TCS **54**, pp. 199–214 (1987)
24. Z. Tang: *Toward a Unified Logical Basis for Programming Languages*. In: R.E.A. Mason, ed.: Proc. IFIP Congress '83. Amsterdam: Elsevier Publishers B.V. (1983)
25. J. D. Ullmann: *Principles of Database and Knowledge-Base Systems*. Computer Science Press (1988)
26. P. Wolper: *Temporal Logic Can Be More Expressive*. Information and Control **56**, pp. 72–99 (1983)