TIME-PARAMETERIZED TEMPORAL LOGIC-BASED FRAMEWORK
 FOR DISCRETE-EVENT SIMULATION

QUAN NGUYEN and TU VAN LE

Faculty of Information Sciences & Engineering
University of Canberra
PO Box 1, Belconnen, ACT 2616, Australia
Email: {quann, vanle}@ise.canberra.edu.au

ABSTRACT

A temporal logic-based language for discrete-event simulation is presented. The language employs
forward reasoning mode and treats time as a first class element of the simulation models. Semantics of
the language's constructs, the simulation procedure are formally discussed. Simulation programs written
from the language generate far more less events than traditional simulation methods.

1. Introduction

Logic programming, since introduced by Kowalski [4], has received enormous attention and has successfully applied
to a wide range of domains. The advantageous features of logic programming are well known: declarative and
procedural reading, symbolic manipulation, powerful pattern matching, built-in theorem prover, to name a few.
However, classical logic is inadequate for modeling applications that contain the notion of dynamic change. To
overcome this limitation, temporal logics [16] are brought in to provide a clean, side-effect free account of time-
dependent properties of certain problems.

Temporal logics are rich, well developed and are suitable for simulation modelling. Several logic programming
languages based on temporal logics have been proposed. (We refer the reader to a comprehensive survey on temporal
and modal logic programming in [13]). Yet, the literature shows very few attempts to build a logic-based language that
specifically caters for system simulation. Even with the extensions of Prolog for the purpose of modelling and
simulation, there are only a handful of them: TS-Prolog [3], LOOPS [14], POSS [6], PFHOS [7]. Recently, a few
temporal logic-based approaches to simulation have been reported. Let us mention Radya [15] who proposed a logic-
based foundation's modelling language that contains many useful temporal constructs but not conforming to the logic
programming style. Liu and Orgun [9] used Chronolog, a general-purpose temporal logic programming language to
write simulation programs, and defined a clocked temporal logic [10] to broaden the simulation domains. Le [8]
provided a fuzzy temporal logic-based framework for fuzzy discrete-event simulation, which forms the basis for
the work presented in this paper. To highlight important aspects of temporal logic approach to simulation, we only present
the "crisp" version of the framework. The notion of fuzzy time can be incorporated into the version with little
difficulty.

2. Motivation

In our opinion, the restriction of using temporal logic programming for simulation modeling primarily stems from
following reasons. First, the main purpose of simulating a dynamic system is to study its behaviour over time, that is,
by starting from a known initial state of the system under study, we wish to find an answer for the question “What will
be the results?” (i.e. forward reasoning). On the other hand, the main theme of logic programming paradigm is goal-
oriented problem solving (i.e. backward reasoning). Thus conceptual difference on reasoning directions might have
discouraged people using logic programming to tackle simulation tasks.

Another important aspect of computer simulation is the treatment of time. A large body of simulation domains
involves explicitly with the physical time. Examples of this type are systems whose states are represented by
differential equations, queuing systems, communication networking, multiprocessor model, job-shop scheduling,
inventory policy, etc. From the simulation's point of view, a series of state transitions may happen at a single time point (e.g., simultaneous events or a state transition triggers another state transition), and vice versa (e.g., state transition may take time). Unfortunately, most of the current temporal logics [13] operate on states (or possible worlds), not on time, and refer to each state as a "time instant" or "moment in time". To overcome the mismatch, one must map each state to a simulation time point. This process is error-prone, and if carried out at programming level, could destroy the declarativeness of the logic program.

The temporal logic version we describe here would tackle the above-mentioned problems. In fact our system employs forward reasoning mode and treats time as a first-class element of the model.

3. Time-parameterized Temporal Logic

In this section, we outline the Time-parameterized Temporal Logic (TPTL), which is the logical foundation of TipLog, a time-parameterized temporal logic-based simulation language. First, we present the notations and terminology used throughout the paper, then we give the model-theoretic semantics of the logic.

3.1 Notation and terminology

Let \( \mathcal{L} \) be a language of TPTL, that is a set of symbols (consisting of variable, constant, function and predicate symbols) and the usual rules of forming formulas [11]. In \( \mathcal{L} \), predicate symbols are classified into dynamic and static predicate symbols. The dynamic predicate symbols consist of event symbols and a special binary predicate symbol 'state'. We will refer to the atomic formulas the predicate symbol of which is 'state' (resp. an event predicate symbol) simply as a state (resp. event). The first argument of 'state' is called state-id and the second one is called state-value.

The intended meaning of those arguments are clear from the terminology, that is, a state-id uniquely identifies an individual state variable of a simulation system, and the associated state-value denotes the value of the corresponding state. For example: \text{state}(queue1, [a,b,c]), \text{state(server}(2), \text{busy}).

Throughout this paper we assume that the time domain \( \mathcal{C} \) is the set of all nonnegative real numbers. We define an internal clock \( \mathcal{E} \) to be a strictly increasing sequence of natural numbers: \(<0,1,2,\ldots>\). Note that the internal clock does not show physical times (i.e., simulation time), but only time marks. Intuitively, it is merely an index of states of computation. In fact, each simulation time point is associated with a (possibly empty) subsequence of \( \mathcal{C} \) as defined below:

**Definition 3.1** A clock-to-time mapping is a non-decreasing function \( \mathcal{C} : \mathcal{E} \rightarrow \mathcal{C} \) which maps each clock's value to a simulation time point. Thus we can divide \( \mathcal{C} \) into subsequences \( s_1,s_2,\ldots,s_n \) such that \( \bigcup \limits_{i=1}^{n} s_i = \mathcal{C} \) and \( \bigcap \limits_{i=1}^{n} s_i = \emptyset \), and each \( s_i \) is associated with a particular time point \( \tau \in \mathcal{C} \). We call \( s_i \) a state transition window associated with \( \tau \).

We also assume that the interpretations of variables, constant, function and ordinary predicate symbols are independent of time as in first-order logic. Only dynamic predicate symbols may have time-variant interpretations.

In TPTL, we define three time-parameterized temporal operators: \( P\alpha, F\alpha, unless(\alpha, \beta) \), where \( \alpha, \beta \) range over \( \mathcal{C} \) and are called temporal variables. Let \( A, B \) be formulas, the informal meaning of \( P\alpha A \) (or \( F\alpha A \)) is "\( A \) is true at \( \alpha \) units of time in the past (or future)" and \( A unless(\alpha, \beta) B \) is "\( A \) is true if \( B \) is not true in the time interval \( [\alpha, \beta] \)". The second argument of the unless operator is called unless precondition. Thus, the formation of formulas can be extended to include the newly defined operators. That is if \( A \) is a state or an event then \( P\alpha A, F\alpha A, (\forall \alpha)P\alpha A, (\exists \alpha)F\alpha A, \) and \( A unless(\alpha, \beta) B \) are formulas. We assume that terms and other formulas are constructed in the traditional way [11], by means of connectives \( \{ \neg, \land, \lor \} \). Let \( E \) be an event, we call \( E \) a positive event and \( \neg E \) a negative event. We now define the notion of interpretation and satisfiability in TPTL.

3.2 Model-theoretic semantics

Let \( D \) be a non-empty set. Let \( f : D \mapsto D \) be a partial function. A state-value association \( \Sigma_f \) is a set of 2-tuples...
The intended interpretation of Σ is an association of state-id's with state-values. Let Σ denote a set of all possible Σ's. We have the following definition:

Definition 3.2.1 A temporal interpretation I of a TPTL-language $\mathcal{L}$ is a tuple $< D, \Sigma, M >$, where $M$ is a meaning function. That is, for each symbol of $\mathcal{L}$, $M$ assigns an element of:

(a) $D$, if the symbol is a constant or variable symbol.
(b) $\mathcal{S}$, if it is a temporal variable.
(c) $[D^k \rightarrow D]$, if it is a $k$-ary function symbol.
(d) $[\mathcal{S} \rightarrow \Sigma]$, if it is the predicate symbol 'state'.
(e) $[\mathcal{S} \rightarrow P(D^k)]$, if it is a $k$-ary event or static predicate symbol, where $P(X)$ denotes the power set of $X$. Note that if $p$ is a $k$-ary static predicate symbol, then $I(p) \in P(D^k)$, which can be regarded as a constant function from $\mathcal{S}$ to $P(D^k)$.

Let $A$ be a formula. We denote the expression that "$A$ is true at clock $c \in \mathcal{S}$ under interpretation I" by $I \models A$. I is called a model of $A$ iff $A$ is true at all values of clock (notation $I \models A$). If $e$ is an term in $\mathcal{L}$, which is constructed by the variable, constant and function symbols in $\mathcal{L}$, then the interpretation $I$ is extended in the usual way to define the value $I(e)$ in $D$.

Definition 3.2.2 Given a temporal interpretation $I$ of TPTL, a clock-to-time mapping $\mathcal{E}$, an event $E$, a state $S$, a (dynamic or static) formula $A$, a dynamic formula $B$, and any $c \in \mathcal{S}$, the semantics of elements of the language $\mathcal{L}$ are defined inductively as follows:

(a) For any $n$-ary predicate symbol $p$, and any terms $e_1, ..., e_n$, we have:
   
   $I \models p(e_1, ..., e_n)$ iff $< p(e_1), ..., p(e_n) > \in I(p)(c).

(b) $I \models \neg A$ iff it is not the case that $I \models A$.
(c) $I \models \forall e \cdot A$ iff $\exists c' : \mathcal{E}(c') = \mathcal{E}(c) - \alpha$ and $I(c') = E$.
(d) $I \models \forall e \cdot S$ iff $\exists c' : \mathcal{E}(c') = \mathcal{E}(c) - \alpha$ and $I(c') = S$.
(e) $I \models \forall e \cdot B$ iff $\exists c' : \mathcal{E}(c') = \mathcal{E}(c) + \alpha$ and $I(c') = B$.
(f) $I \models A \text{ unless } (\alpha, \beta) A_2$ iff $\mathcal{E}(c) \geq \beta$ and either $(\forall e' : \mathcal{E}(c') = [\alpha, \beta])$, $I, \text{ min}(c,c') = \neg A_2$ and $I \models A_2$ or $(\exists e' : \mathcal{E}(c') = [\alpha, \beta])$, $I, \text{ min}(c,c') = A_2$.
(g) $I \models A \land A_2$ iff $I \models A_2$ and $I \models A_2$.
(h) $I \models A_1 \rightarrow A_2$ iff from $I \models A_1$, then $\exists c' > c$ and $\mathcal{E}(c') = \mathcal{E}(c)$ such that $I(c') = A_2$.
(i) $I \models \forall (x) A$ iff $I[d/x], c = A$ for all $d \in D$ (or $I \models true$ if $x$ is a temporal variable).

We have the following inference rule which can be proved from (h) above ($A(c)$ denotes we have $A$ at $c$):

\[
\frac{A(c) \quad (A \rightarrow B)(c)}{B(c+k) \quad \text{where } k > 0 \text{ and } \mathcal{E}(c+k) = \mathcal{E}(c) \text{ for some } k.}
\]

The inference rule ensures that the consequence of an implication will be evaluated to true at the same state transition window (i.e. same simulation time instant) with the antecedent in a nondeterministic way. Note also that in (c) and (e) an event is considered to be occurred at a time point $\alpha$ iff it bounds to occur at any state in the state transition window of $\alpha$ (def. 3.1). Also, (d) implies the value of a state in the past time, say $\alpha$, is only retrieved from the last state of the state transition window of $\alpha$. This is natural if we think that the sequence of all state changes at a particular time point is a series of independent actions of simulation components which produces a global state transition as a net result [17].
4. Syntax of TipLog

*TipLog* is a simulation language that is based on a fragment of *TPTL* in the sense that not all possible formations of semantics constructs are allowed. Thus the language’s interpreter is simpler and more efficient. It turns out that the following *TipLog*’s program syntax is adequate for expressing most of dynamic aspects of simulation modeling.

Let \( A, B, E, \) and \( S \) denote any static predicate, dynamic predicate, event and state respectively. Let \( \alpha, \beta \) and \( \delta \) be temporal variables. Note that \( B = PaB = PxB \) if \( \alpha = 0 \). We denote by *Initialize*, a special event scheduled at the time 0. The formulas in *TipLog* are formed by the ways described below:

<table>
<thead>
<tr>
<th>Static Head: ( SH )</th>
<th>::= ( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Body: ( SB )</td>
<td>::= ( \emptyset ) | ( A ) | ( SB_1 \land SB_2 )</td>
</tr>
<tr>
<td>Antecedent: ( Ant )</td>
<td>::= ( E ) | ( PaS ) | ( \neg Ant ) | ( Ant_1 \land Ant_2 )</td>
</tr>
<tr>
<td>Consequence: ( Con )</td>
<td>::= ( \emptyset ) | ( PaB ) | ( F_{\alpha}B ) | ( unless(\beta, \delta) B_2 ) | ( Con_1 \land Con_2 )</td>
</tr>
<tr>
<td>Initial Clause: ( Init )</td>
<td>::= <em>Initialize</em> ( \rightarrow ) ( SB \land Con )</td>
</tr>
<tr>
<td>Static Clause: ( SC )</td>
<td>::= ( SB ) ( \rightarrow ) ( SH )</td>
</tr>
<tr>
<td>Dynamic Clause: ( DC )</td>
<td>::= ( Ant ) ( \rightarrow ) ( SB \land Con )</td>
</tr>
</tbody>
</table>

A *TipLog* program is a finite set of clauses including an initial clause. For notational convenience, we adopt Prolog style in using ‘;’ standing for ‘\( \land \)’.

5. Execution of a *TipLog*-simulation program

**Definition 5.1** Let \( \mathcal{P} \) be a *TipLog*-simulation program.

(a) An *event history* is a list \( \mathcal{E} = [c_1; LE_1, \ldots, c_n; LE_n] \), where for each \( i \), \( c_i \in \mathcal{E} \), and \( LE_i \) is a list of event occurrences at clock \( c_i \).

(b) A *state history* is a list \( \mathcal{S} = [c_1; LS_1, c_2; LS_2, \ldots, c_n; LS_n] \), where for each \( i \), \( c_i \in \mathcal{E} \), and \( LS_i \) is of the form \([(id_{i1}, val_{i1}), \ldots, (id_{ik}, val_{ik})]\), which is a list of state-id and state-value pairs that are considered to be true at clock \( c_i \).

(c) A *running set* \( RS \) is a collection of event and state predicates that are yet to be resolved. More formally, \( RS = \{ T_i; F_i \mid i \in N \} \), where \( T_i \in \mathcal{E} \) and is called *time-tag*, and \( F_i \) is either a predicate \( A \) or has the form \( (A_i, unless(\alpha, \beta) A_j) \). Here the formula \( F_i \) is said to be *temporally grounded* (notation \( \downarrow T_i; F_i \)) if \( T_i \) is a ground term (i.e. a real number); otherwise \( F_i \) is said to be *temporally ungrounded* (notation \( \uparrow T_i; F_i \)).

(d) A *\( \theta \)-global substitution* is a substitution of the variables in \( \mathcal{E}, \mathcal{S} \) and \( RS \) with the values given in \( \theta \) (variable assignment is defined in the traditional way).

(e) A *run environment* \( \mathcal{R} \) is a 5-tuple \( < RS, \theta, \mathcal{C}, Clock, Time > \), where \( RS \) is a running set, \( \theta \) a global substitution, \( \mathcal{C} \) a clock-to-time mapping, \( Clock \in \mathcal{E} \), and \( Time \in \mathcal{R} \).

(f) A *simulation result* is a 3-tuple \( I_{Result} = < D, \Sigma, M > \).

The purpose of a simulation run is to find a model of the program \( \mathcal{P} \), that is, an interpretation which satisfies the semantics of the program clauses in \( \mathcal{P} \). As a side-effect, a clock-to-time function \( \mathcal{C} \) is to be determined which provides a mapping from the states of the program execution to the simulation time points.
Let $E_i$ denote an event, $S_i$ a state, $G_i$ either an event or a state respectively. Also let $DC$ be a dynamic clause, $Ant$ an antecedent, $Con$ a consequence, $\sigma$ a variable substitution. Let $SSet(F)$ denote the set of states (possibly containing the past operator or negation connective) appearing in the formula $F$. $ESet(F)$ the set of positive events appearing in $F$, and $\overline{ESet(F)}$ the set of negative events appearing in $F$ (i.e. the negation of events). We denote the state-id of state $S$ by Id($S$) and the state-value by Val($S$).

**Algorithm 5.1 (Simulation Procedure)**

Set $Clock$, Time, and $\mathcal{E}(0)$ to 0
Set lists $\mathcal{E}$ and $\mathcal{S}$ to [].
Set $\theta$ to $\emptyset$.
Set running set $RS$ to $\{0:\text{Initialize}\}$

**Repeat**

Execute events (Algorithm 5.2)

Advance Time (Algorithm 5.4)

**Until** Time $>$ the specified simulation time or $RS = \emptyset$ or error encountered

If Run successful then

Build $I_{\text{result}}$ (Algorithm 5.5)

**Algorithm 5.2 (Execute Events)**

**Repeat**

Let $Q_k = \{E_i \mid \text{Time}: E_i \in RS\}$

If $Q_k \neq \emptyset$ then

Let $K = \emptyset$

For each antecedent $Ant$ in $\mathcal{P}$

If $\exists \sigma : ESet(Ant)(\sigma) \subseteq Q_k(\sigma)$, and $ESet(Ant)(\sigma) \cap Q_k(\sigma) = \emptyset$, and $S$ is true $\forall S \in SSet(Ant)(\sigma)$ (by scanning the state history) then

Resolve all static predicates in $Con(\sigma)$ resulting a variable substitution $\beta$, let $\lambda = \sigma\beta$

Perform $\lambda$-global substitution on $RS$, $K$, $Q_k$, event history and state history

For each dynamic atom $RxG$ in $Con$ do

Insert $(\alpha + \text{Time}):G$ to $RS$ ($G$ may include unless operator and $\alpha$ can be 0 or ungrounded)

End For

Let $K = K \cup ESet(Ant)(\lambda)$

Set $\theta$ to $\theta\lambda$

End If

End For

If $K = \emptyset$ then Report error and stop run

For each event $E_k$ in $K$

Insert $Clock:E_k$ to the event history

Remove $E_k$ from $Q_k$ and $RS$

End for

Set $\mathcal{E}(\text{Clock})$ to $Time$

Set $Clock$ to $Clock + 1$

End If

Execute internal transitions (Algorithm 5.3)

Let $Q_k = \{S_i \mid \text{Time}: S_i \in RS\}$

Until $Q_k \neq \emptyset$ and $Q_k = \emptyset$
Algorithm 5.3 (Execute Internal Transitions)

Let $Q_5 = \{ S_i | \text{Time}: S_i \in RS \}$
If $Q_5 = \emptyset$ then exit the algorithm
Let $K = \emptyset$
For each antecedent $Ant$ in $\mathcal{S}$
  If $\exists \sigma: (Q_5(\sigma) \cap \mathcal{S}et(Ant)(\sigma)) \neq \emptyset$, and $\mathcal{E}set(Ant)(\sigma) = \emptyset$, and $\mathcal{E}SET(Ant)(\sigma) \cap Q_5(\sigma) = \emptyset$,
  and $S$ is true $\forall S \in \mathcal{S}et(Ant)(\sigma) \setminus S_i(\sigma)$ (by scanning the state history) then
  Resolve all static predicates in $Con(\sigma)$ resulting a variable substitution $\beta$, let $\lambda = \sigma\beta$
  Perform $\lambda$-global substitution on $RS$, $Q_5$, event history and state history
  For each dynamic atom $F \& G$ in $Con$ do
    Insert $(\alpha + \text{Time}):(G \to RS$ ($G$ may include unless operator and $\alpha$ can be 0 or ungrounded)
  End For
  Set $\theta$ to $\theta\lambda$
End If
End for

If in $Q_5$, there exists a pair $<S_m, S_n>$ such that $Id(S_m) = Id(S_n)$ and $\text{Val}(S_m) \neq \text{Val}(S_n)$ then
states are in conflict, report error and stop run
For each $S_k$ in $Q_5$
  Insert $Clock: <Id(S_k), \text{Val}(S_k)>$ to the state history
  Remove $S_k$ from $RS$
End for
Set $\exists Clock$ to $\text{Time}$
Set $Clock$ to $Clock + 1$

Algorithm 5.4 (Advance Time)

If $\exists T_i; E_i \in RS: T_i \leq \text{Time}$ then
  Report error and stop the run.
Let $Q = \{ T_i : \downarrow T_i; G_i \in RS \text{ and } T_i > \text{Time} \}$ (Note that $RS$ may contain temporally ungrounded formulas and in $Q$ we only select temporally grounded formulas)
If $Q = \emptyset$ then
  Report error and stop the run.
Let $\text{NextTime} = \text{Min} Q$
For each formula $F$ of the form $\text{NextTime}: G$, unless($\alpha, \beta$) $G_j$ do
  If either $\alpha$, $\beta$ are ungrounded or $\alpha > \beta$ or $\beta > \text{NextTime}$ then
    Report error and stop run
Else
  If $\exists \sigma: G_i(\sigma)$ is true in at least one point in the interval $[\alpha, \beta]$ (by scanning $\mathcal{E}(\sigma)$ or $\mathcal{E}(\sigma)$) then
    Perform $\sigma$-global substitution on $RS$, event history and state history
    Set $\theta$ to $\theta\sigma$
  Else
    Insert $\text{NextTime}: G_i$ to $RS$
  End If
End If
Discard $F$ from $RS$
End For
Set $\text{Time}$ to $\text{NextTime}$
Algorithm 5.5 (Build $I_{Result}$)

Given the run environment $\mathcal{R} = <RS, \theta, \mathcal{E}, \text{Clock}, \text{Time}>$, and the event list $\mathcal{E}$, the state history list $\mathcal{O}$, we build the meaning function $M$ as follows:

(a) Each variable and each temporal variable is given its assignment according to $\theta$.
(b) Each constant, function and static predicate symbol is given its assignment in the classical way (i.e. Herbrand model).
(c) Let $M_E$ be the function that $M$ assigns to the event predicate symbol $E$. We construct $M_E$ as follows:

For each $i$ from 0 to Clock do
- In $\mathcal{E}$, find $c_E = i$, if found then
  - Set $M_E(i) = \text{elements in } LE_E$, the predicate symbol of which is $E$.
- Else
  - Set $M_E(i) = \emptyset$

(d) Let $M_S$ be the function that $M$ assigns to the predicate ‘state’. We construct $M_S$ as follows:

- Set $M_S(0) = LS_0$ in $\mathcal{O}$
- For each $i$ from 1 to Clock do
  - In $\mathcal{O}$, find $c_S = i$, if found then
    - Set $L = \text{elements in } LE_i$
  - Else
    - Set $L = \emptyset$
  - End If
- Set $M_S(i) = M_S(i-1) \cup (L \setminus M_S(i-1))$

Let $\Sigma = \{ M_S(i) \mid i \in 0, 1, ..., \text{Clock} \}$, then we have $I_{Result} = <D, \Sigma, M>$.

6. Soundness of TipLog-simulation Procedure

Theorem 6.1 If $\mathcal{P}$ is a TipLog-simulation program and $I_{Result}$ is the simulation result, then $I_{Result}$ is model of $\mathcal{P}$, that is:

$$I_{Result} = \mathcal{P}$$

7. Programming in TipLog

Example 7.1 Following is the TipLog-simulation program for a single-server queueing system (M/M/1). Here, interarrival and service_time are static predicates, arrival, departure, queue are events. Also the $s, q$ represent the identifier of server and queue respectively. The following program clauses are organized in a common approach found in most simulation languages. We write $(i + 1)$ to denote $I_i$ where $I_i$ is $i + 1$ as in Prolog.

initialize $\rightarrow$ state($q$, []), state($s$, idle), interarrival($T$), $F(T)$:arrival(1). [Initialize]

arrival($I$), state($q$, $L$) $\rightarrow$ append($L$, [], $I_1$), state($q$, $L_1$), interarrival($T$), queue($I$), $F(T)$:arrival($I_1$+1). [clause 1A]

queue($I$), state($q$, []) $\rightarrow$ state($q$, []), state($s$, busy), service_time($T$), $F(T)$:departure($I_1$). [clause 2A]

queue($I$), state($q$, [H | $T$]) $\rightarrow$ append([H | $T$], [], $I_1$), state($q$, $L$). [clause 3A]
departure(I), state(s, [I]) → state(q, []), state(s, free). \[\text{clause 4A}\]

departure(I), state(q, [I, H | L]) → state(q, [H | L]), service_time(T), F(T):departure(I+1). \[\text{clause 5A}\]

Program 7.1: A common approach to M/M/1

Example 7.2 TipLog allows us to write a more compact and elegant simulation program (we assume the same initial clause as in example 7.1):

arrival(I), state(q, L) → append(L, [I], L1), state(q, L1), interarrival(T), F(T):arrival(I+1). \[\text{clause 1B}\]

state(s, free), state(q, [H | L]) → state(q, L), state(s, busy), service_time(T), F(T):state(s, free). \[\text{clause 2B}\]

Program 7.2: TipLog approach to M/M/1

8. Concluding Remarks

In what follows, we highlight some advantageous features of TipLog by comparing the programs 7.1 and 7.2.

It is clear that program 7.2 generates far fewer number of events than program 7.1 by allowing the so-called internal state transition [17], which generates an implicit event if the system’s state satisfies some predefined conditions (clause 2B) to bring the system to a new state. In the same way, we can model contingent events, as defined by Nance [12], the occurrence of which can be expressed in a way which is not strictly as a function of system time (e.g. dependent on the system state). It is important to note that, TipLog does not evaluate the conditions of contingent events every time the system’s state changes as usually done in most simulation languages (i.e. busy waiting)[1]. A more efficient way is to check the conditions only if some state changes may affect the truth of the conditions. This is usually done via pointers or by using an additional list to store the monitored state variables. TipLog offers a natural and logical approach to solve this problem by treating a state change as a “quiet” event. This can be revealed by studying the TipLog’s simulation procedure.

Note also that the clauses in program 7.1 are organized so that one clause triggers another by means of an event. Furthermore, in this program one must take into account all special cases and arrange actions accordingly. For instance, if a queue event occurs while the current waiting queue is empty then the event wakes up the server, otherwise the arriving entity will join the queue (clause 2A and 3A). Another example is that upon a departure, the server’s status will become free if the queue is empty, otherwise the server’s status stays busy as it will serve for the next member of the queue (clause 4A and 5A). Program 7.2 shows that we need not attend to such details. The arriving entity just simply joins the queue regardless of the current server’s status. On the other hand, the server keeps taking elements from the queue as long as there is one. This is similar to the classic producer-consumer problem. Here, however, there is no need to arrange wake up operations nor to engage in busy-waiting mode. Therefore, the approach enhances the program modularity in a natural and logical way.

Due to space limitation, the features listed below have not been illustrated in this paper, but we conjecture that they are within the scope of TipLog:

- TipLog can be used to simulate certain combined discrete-continuous models.
- The ability to handle simultaneous events (e.g. multiple events and negative events may be specified in an antecedent).
- The logical use of the unless operator for cancelling a pre-scheduled event or state change, and for modeling interruption and process preemption.
- The facilitation of process communication through the use of shared variables between predicates.
- The provision of coroutine processing capability for the process world view (predicates are resolved by time order, not by program clauses’ order). A process can be suspended and re-activated through the use of future time variables.
• The ability to model all three standard simulation world views (i.e. event, activity and process) [2].
• The freedom from the frame problem in logic programming [5], and the absence of destructive assignment. Thus the declarativeness of updating system’s states is achieved.

References


