Fluent Temporal Logic for Discrete-Time Event-Based Models

Emmanuel Letier
Dpt. d'Ingénierie Informatique
Université Catholique de Louvain
Louvain-la-Neuve, Belgium
eletier@info.ucl.ac.be

Jeff Kramer, Jeff Magee, Sebastian Uchitel
Department of Computing
Imperial College London
and London Software Systems, U.K.
{jk, jnm, s.uchitel}@doc.ic.ac.uk

ABSTRACT

Fluent model checking is an automated technique for verifying that an event-based operational model satisfies some state-based declarative properties. The link between the event-based and state-based formalisms is defined through "fluents" which are state predicates whose value are determined by the occurrences of initiating and terminating events that make the fluents values become true or false, respectively.

The existing fluent temporal logic is convenient for reasoning about untimed event-based models but difficult to use for timed models. The paper extends fluent temporal logic with temporal operators for modelling timed properties of discrete-time event-based models. It presents two approaches that differ on whether the properties model the system state after the occurrence of each event or at a fixed time rate. Model checking of timed properties is made possible by translating them into the existing untimed framework.

Categories and Subject Descriptors
D.2.1 [**Software Engineering**]: Requirements/Specifications, Software/Program Verification – *model checking*

General Terms
Theory, Verification.

Keywords
Fluent linear temporal logic, discrete-time event-based models, model-checking, software architecture analysis.

1. INTRODUCTION

Event-based models are convenient formalisms for modelling behaviours of complex systems at the architectural level. They describe a system as a set of interacting components where each component is modelled as a state machine and interactions between components occur through shared events corresponding to the messages sent and received by components or to service invocations initiated or accepted by components. Such models provide the basis for a wide range of automated analysis techniques, notably deadlock analysis, model animation and model verification through model checking. For example, the Architectural Description Language (ADL) Wright [1] is based on the event-based process algebra CSP [7] for behaviour descriptions and FDR [23] for behaviour analysis; PADL [2] is based on CCS [20] and via TwoTowers, uses the Concurrency Workbench [3] for functional analysis; our own Darwin ADL [16] uses the process algebra style language FSP, and the Labeled Transition System Analyzer (LTSA) tool [17].

For model verification, the properties to be satisfied by the architecture are typically expressed in some form of temporal logic. Specifying these properties is often much easier if they can refer to system states in addition to referring to events. This motivated the use of "fluents" to provide a uniform framework for specifying properties that combine event and state predicates and for automatically verifying the satisfaction of those properties by an event-based model [5]. A fluent is a state predicate whose value is determined by the occurrences of initiating and terminating events that make the fluent value become true or false, respectively.

Event-based modelling languages such as CSP, CCS and FSP are untimed. Their semantics are sequences of events specifying the order in which events occur, but not the actual time at which events occur or the delay elapsed between consecutive events. A standard way to model time in these untimed formalisms is to include an explicit tick event signalling the successive ticks of a global clock to which each timed component synchronizes [23, 17]. These models are called discrete-time event-based models because the resulting time domain is isomorphic to the natural numbers.

Fluent linear temporal logic (FLTL) defined in [5] is convenient for specifying requirements on untimed event-based models. It is however very difficult to use for specifying requirements on timed event-based systems because it involves writing complex formulae with explicit references to clock events.

Various extensions of temporal logic allow one to naturally model properties of timed systems using bounded versions of temporal logic operators [12, 6]. For example, the bounded eventually operator $<>_{\leq d}$ P means that P holds at some future time within the next d time units. However these extensions are defined for state-based models only and cannot be used to conveniently model properties of timed event-based models.

The objective of this paper is to extend FLTL with bounded versions of temporal logic operators for modelling timed properties of discrete-time event-based models. The paper describes two approaches.

In the first approach, we consider properties describing sequences of system states observed after each occurrence of an event, as in standard FLTL. The time of the system in a given state is counted by the number of tick events that have occurred since the beginning of the execution. Bounded versions of temporal logic operators such as $<>_{\leq d}$ P are defined accordingly. We define an encoding of the bounded operators into untimed FLTL assertions so that bounded FLTL assertions can be model checked in the existing untimed framework.

In the second approach, we consider properties describing sequences of system states observed at a fixed time rate, instead of after each occurrence of an event. In this case, zero, one or more events may occur between two consecutive states and the time of the system in a state is counted by the position of the state in the sequence. In the context of this paper, temporal logics used to describe sequences of states observed at a fixed time rate will be called *synchronous temporal logics*, while those used to describe sequence of states observed after each occurrence of an event will be called *asynchronous*. Note that modelling system properties in a synchronous or asynchronous temporal logic does not necessarily mean that the event-based model being verified is also synchronous or asynchronous. (An event-based model is asynchronous if it models components that execute at arbitrary relative speeds and it is synchronous if it models components that execute in lockstep [20]). Synchronous temporal logic is a natural approach for describing properties of discrete-time *state-based* models [6]. It is used in particular in the KAOS goal-oriented requirements engineering method for the formal specification of system goals and requirements [14]. We will see that there exist subtle differences between the interpretations of the temporal logic operators in the two approaches. The difference applies to standard temporal operators as well as bounded ones. This may be an important source of confusions and errors for modellers that incorrectly use assertions defined in one formalism with a tool based on the other formalism. Our aim is to clarify the differences between the two existing paradigms and to define an encoding of synchronous temporal logic into asynchronous FLTL so that synchronous assertions can be model-checked in LTSA.

The paper is organized as follows. Section 2 presents the necessary background on temporal logic, labelled transition systems and fluent temporal logic. Section 3 extends asynchronous fluent temporal logic with bounded temporal operators and defines an encoding of these operators into untimed FLTL. Section 4 illustrates the use of these operators on the light control problem. Section 5 defines synchronous fluent temporal logic and defines a mapping from synchronous assertions into asynchronous ones. Section 6 illustrates model checking of synchronous fluent temporal logic assertions on the mine pump case study.

2. BACKGROUND

2.1 Linear Temporal Logic

Given a set of atomic propositions Π, a well-formed LTL formula is defined inductively using the standard Boolean operators, and the temporal operators X (next), [] (always), <> (eventually), U (until) and W (Awaits) as follows:

- each member of Π is a formula
- if P and Q are formulas, then so are $\neg$P, P$\wedge$Q, P$\vee$Q, P $\rightarrow$ Q, P $\leftrightarrow$ Q, X P, [] P, <> P, P U Q, P W Q.

An interpretation for an LTL formula is a infinite trace *h: Nat ->* 2^{Π} that maps to each position $i \in$ *Nat* the set of propositions that hold at that position. The notation (h,i) |= P is used to express that the LTL formula P is true at position i of the trace h. The semantics of the temporal logic operators is then defined as follows [19]:

- (h,i) |= X P *iff* (h, i+1) |=P
- (h,i) |= [] P *iff* (h, j) |=P *for all* j $\geq$ i
- (h,i) |= <> P *iff* (h, j) |=P *for some* j $\geq$ i
- (h,i) |= P *U* Q *iff* (h, j) |= Q *for some* j $\geq$ i *and* (h, k) |=P *for all k s.t.* i $\leq$ k < j
- (h,i) |= P *W* Q *iff* (h,i) |= P *U* Q *or* (h,i) |= []P

The Boolean operators have their usual semantics. A LTL formula P is said to be satisfied by a trace h, noted h |= P, if it is satisfied at the initial position, i.e. (h,0) |= P.

2.2 Metric Temporal Logic

Metric Temporal Logic (MTL) extends LTL with the following bounded temporal operators $[]_{\sim d}$ P , $<>_{\sim d}$ P and P $U_{\sim d}$ Q where $\sim \in \{<, \leq, >, \geq\}$ and d $\in$ Nat [12, 6].

The semantics of these bounded operators is defined over infinite traces *h: Nat ->* 2^{Π} enriched with a time domain *T* and a temporal distance function

dist: Nat $\times$ *Nat -> T*

where *dist(i,j)* denotes the time elapsed between positions i and j in a trace. Different choices of temporal domain and distance function are possible as long as they satisfy all desired properties of a metric [12].

A common choice of temporal distance consists in considering traces in which consecutive states are always separated by a single time unit [6]. The distance function is therefore defined as *dist(i,j)* = $|j-i| \times \delta$ where δ is the arbitrarily chosen time unit. The time domain is the natural multiple of δ and is therefore discrete. In this paper, a temporal logic with this choice of temporal distance function is called a *synchronous* temporal logic because it describes traces in which the system state is observed at a fixed time rate.

Another common temporal distance function consists in admitting arbitrary and varying real-numbered delays between consecutive states [6]. This results in a continuous model of time. This model is more expressive but leads to difficulties in automated analysis. Dense-time models will not be considered in this paper.

The semantics of the bounded temporal operators is defined over system traces and distance functions as follows:

- (h,i)|= $[]_{\sim d}$ P *iff* (h, j)|=P *for all* j $\geq$ i and dist(i,j) ~ d
- (h,i)|= $<>_{\sim d}$ P *iff* (h, j)|=P *for some* j $\geq$ i and dist(i,j) ~ d
- (h,i)|= P $U_{\sim d}$ Q *iff* (h, j)|=Q *for some* j $\geq$ i and dist(i,j)~ d and (h, k) |=P for all k s.t. i $\leq$ k < j

Note that for synchronous LTL, where next time and next state are semantically equivalent, the bounded temporal operators do not add expressive power to standard unbounded LTL operators because they can be expanded into LTL formulae involving the next state operator. For example, the assertion $<>_{\leq 3}$ P is equivalent to (P $\vee$ X P $\vee$ X X P $\vee$ X X X P) and the assertion $[]_{\leq 3}$ P is equivalent to (P $\wedge$ X P $\wedge$ X X P $\wedge$ X X X P). These operators however add succinctness and ease of use to LTL for describing bounded temporal requirements.

The next state operator X of MTL is the standard X operator of LTL defined in Section 2.1. The above equivalences hold for synchronous LTL only, which is the only timed extension of LTL in which the 'next state' operator X also means 'at the next time unit'. With other choices of temporal distance function, the X operator no longer corresponds to the next time unit.

2.3 Labelled Transition Systems

We use Labelled Transitions Systems (LTS) to model the behaviour of interacting components [17]. An LTS model describes a system as a set of concurrent components where each component is characterized by a set of states and by the possible transitions between these states where each transition is labelled by an event. The global system behaviour is the result of the parallel composition of each component LTS so that the components execute asynchronously but synchronize on shared events.

Let *Act* be the universal set of observable events and let τ denote a local action that is unobservable by a component's environment. An LTS *M* is a quadruple $<Q, A, \delta, q_0>$ where:

- Q is a finite set of states,
- $A \subseteq Act$ is the communicating *alphabet* of *M*,
- $\delta \subseteq Q \times A \cup \{\tau\} \times Q$ is a labelled transition relation,
- $q_0 \in Q$ is the initial state.

The semantics of an LTS *M* is a set of sequences of events (observable or τ) that the LTS can perform starting in its initial state.

The parallel composition operator "||" is a commutative and associative operator that combines the behaviour of two LTSs by synchronizing the events common to their alphabets and interleaving the remaining events.

Discrete-time systems can be modelled as LTS by including explicit tick events signalling the regular ticks of a global clock to which each timed processes synchronizes [17].

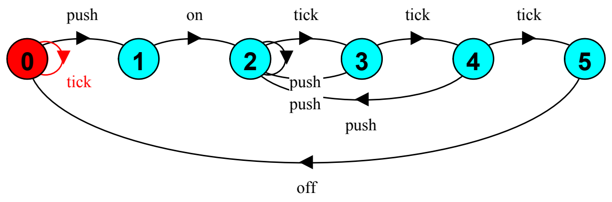

Figure 1. LTS Model of a light with a timer

Finite State Processes (FSP) is the input notation for the LTSA tool. It is a simple process algebra used as a concise way to specify LTS. For example, a light that turns itself off automatically after 3 time units may be specified by the following FSP model:

```
TimedLight = Off,
      Off = ( push -> on -> On[3]  | tick -> Off),
      On[d:0..3] = ( when (d==0) off -> Off
                        | when !(d==0) tick -> On[d-1]
                        | when !(d==0) push -> On[3]).
```

In the above, "->" denotes action prefix, "|" choice and the "when" clause is used to express conditional choice. The symbol ! denotes logical negation. The event push denotes the pushing of the light button, the events on and off denotes the actual turning on and off of the light. The LTS that corresponds to this FSP process is depicted in Fig 1.

2.4 Fluent Linear Temporal Logic

Fluent linear temporal logic (FLTL) is a formalism for specifying state-based temporal logic properties over an event-based operational model.

A *fluent Fl* is a proposition defined by a pair of sets, a set of initiating events $Init_{Fl}$ and a set of terminating events $Term_{Fl}$, and by an initial value $Initially_{Fl}$ that can be true or false. The sets of initiating and terminating events must be disjoint. The concrete syntax for fluents in LTSA is the following:

fluent Fl = < $Init_{Fl}$, $Term_{Fl}$ > initially $Initially_{Fl}$

By default, the initial value of a fluent is false.

A well-formed FLTL formula is an LTL formula whose atomic propositions are fluents.

A set Φ of fluents defines a mapping from event-based to state-based traces. Let $tr: Nat \rightarrow A$ be an event trace, the corresponding state-based trace $h = StateTrace(tr)$ is defined as follows: for every position $i \in Nat$ and every fluent $Fl \in \Phi$, *Fl* is true at position *i* of *h* iff either of the following conditions holds

(a) *Fl* holds initially and no terminating event has occurred before position *i*:

$Initially_{FL}$ *and there is no* $k \in Nat,\ 0 \le k \le i$ *s.t.* $tr(k) \in Term_{Fl}$

(b) some initiating event has occurred before position *i* and no terminating event has occurred since then:

there is some $j \in Nat, j \le i,$ *s.t.* $tr(j) \in Init_{Fl}$
and there is no $k \in Nat, j < k \le i$ *, s.t.* $tr(k) \in Term_{Fl}$

A FLTL assertion *P* is said to be satisfied by an event trace *tr*, noted $tr \models P$, iff $StateTrace(tr) \models P$.

Note that the interval over which a fluent holds is *closed* on the left and *open* on the right. This means that an event occurring at position i of an event-based trace has an effect on the values of fluents at position i of the associated state-based trace. This is slightly different from the fluents of Miller and Shanahan [21] that hold over intervals that are *open* on the left and *closed* on the right which means that an event occurring at position i of an event-based trace has an effect on the values of fluents at position i+1 of the associated state-based trace.

FLTL assertions can also refer to event occurrences. For every event *e* in an LTS model there is an implicit fluent, also noted *e*, whose set of initiating events is the singleton event {*e*} and whose set of terminating events contains all other events in the system alphabet:

fluent e = <e, A-{e}> initially false

According to this definition, the fluent associated with an event *e* becomes true the instant *e* occurs and become false with the first occurrence of a different event.

The concrete syntax for FLTL formulas used in LTSA follows as closely as possible the LTL syntax used in SPIN. In particular, the

ASCII expressions !, &&, and || are used to denote logical negation, conjunction and disjunction, respectively.

Examples. For the light model, a property requiring the light to eventually be on after a tick event may be specified as follows:

```
[](push -> <> LightOn)
```

where LightOn is a fluent defined as follows:

```
fluent LightOn = <on, off>
```

This property is of course very weak because it does not require the light to be turned on immediately after a push event. The correct specification of the required property in FLTL is not straightforward. The assertion

```
[](push -> LightOn)
```

is too strong because it forbids a push event from occurring when the light is off. Modelling the property as

```
[](push -> X LightOn)
```

should be avoided because the resulting formula is not closed under stuttering [13]: the satisfaction of this property by an event trace is affected by the insertion or removal of unobservable τ events. A correct way to model the required property in FLTL is:

```
[](push -> (!tick W LightOn))
```

This assertion requires that after every occurrence of a push event, the light must be on before the next occurrence of a tick event.

As a second example, the property requiring the light to be eventually turned off when no further push events occur may be specified as follows:

```
[] ((! push W ! LightOn) -> <> ! LightOn)
```

This property is again too weak because it does not require the light to be turned off within 3 time units. Specifying such bounded properties in FLTL is extremely hard. The following section will define bounded temporal logic operators facilitating the specification of such properties.

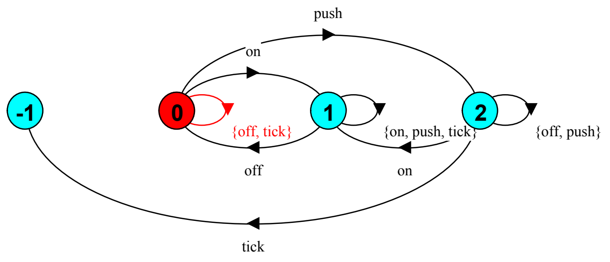

Figure 2. Property LTS for [](push -> (!tick W LightOn))

A technique for model-checking the satisfaction of an FLTL assertion ϕ by an LTS is described in [5]. As for standard model-checking, the general idea consists in generating a Büchi automaton B that recognizes all infinite event-based traces that violate ϕ and check that the synchronous product of B with the LTS is empty. When the property to be verified is a safety property, the Buchi automaton can be viewed as a "property LTS", i.e. a LTS with an ERROR state so that executions leading to the ERROR state correspond to undesired system behaviours. Model checking safety properties therefore reduces to a simple reachability search for the ERROR state. For example, Figure 2 shows the property-LTS generated by the tool for the safety property [](push -> (!tick W LightOn)). In LTSA, the ERROR state is noted by -1.

3. ASYNCHRONOUS DISCRETE-TIME FLTL

This section defines a metric temporal logic that extends FLTL with bounded temporal logic operators. Section 3.1 defines the semantics of these operators by defining the temporal distance appropriate to timed event-based models. Section 3.2 defines how to model check properties involving bounded temporal operators by translating them into untimed FLTL assertions with explicit references to tick events.

3.1 Semantics of Bounded FLTL Operators

In order to define the semantics of bounded operators for FLTL, we need to provide a temporal distance function on the positions of state-based traces.

Let h be a state-based trace. The time elapsed between the positions i and j of h is given by the number of times a tick event occurs between i and j:

$$dist(i,j) = \# \{k \in (i, j] \mid (h,k) \models tick \}$$

Note that the interval in this definition is open on the left and closed on the right. This means that if a tick event occurs in position i and the next tick event occurs in position n, the temporal distance between i and j is nill for all j bigger or equal to i and strictly smaller than n.

The semantics of bounded FLTL operators is then defined as in Section2.2. For example, the assertion "$<>_{\leq 3}$ P" means that P holds at some future time position that is separated from the current position by no more than 3 ticks.

Examples of timed properties for the timed light model are the following:

$[]\ (push \to <>_{<1} LightOn)$
$[]\ (on \to []_{<3} LightOn)$
$[](on \to <>_{\leq 3} (off \vee push))$

The first property requires that when a push event occurs, the light must be turned on within the same time unit. The second property requires that once the light has been turned on, it must remain on during the next 3 time units. The third property requires that when the light is turned on, it be eventually turned off within 3 time units except if a push event occurs during that time.

The concrete ASCII syntax for bounded temporal operators in the LTSA consists in writing the temporal bound into braces as follows <>{<=d} P.

3.2 Model-Checking Bounded FLTL Assertions

In order to model check FLTL assertion involving bounded temporal operators, these operators are translated into unbounded FLTL assertions involving explicit references to tick events. The translation rules are given by the recursive function *Tr* defined as follows:

$$\begin{array}{lcll} Tr([]_{<d}\ P) & = & (P\ W\ (tick \wedge P)) & \text{if } d{=}1 \\ & & (P\ W\ (tick \wedge P \wedge X\ Tr([]_{<d-1}\ P))) & \text{if } d{>}1 \\ Tr([]_{\leq d}\ P) & = & Tr([]_{<d+1}\ P) & \end{array}$$

$$\begin{array}{lll}
Tr(\langle\rangle_{<d}\ P) = & (!\ tick\ W\ P) & \text{if } d=1 \\
 & ((!\ tick \vee X\ Tr(\langle\rangle_{<d-1}\ P))\ W\ P) & \text{if } d>1 \\
Tr(\langle\rangle_{\le d}\ P) = & Tr(\langle\rangle_{<d+1}\ P) & \\
Tr([]_{\ge d}\ P)\ = & []\ P & \text{if } d=0 \\
 & (!tick\ W\ []\ P) & \text{if } d=1 \\
 & (!tick\ W\ X\ Tr([]_{\ge d-1}\ P)) & \text{if } d>1 \\
Tr([]_{<d}\ P)\ = & Tr([]_{\ge d-1}\ P) & \\
Tr(\langle\rangle_{\ge d}\ P) = & \langle\rangle\ P & \text{if } d=0 \\
 & \langle\rangle(tick \wedge \langle\rangle\ P) & \text{if } d=1 \\
 & \langle\rangle(tick \wedge X\ \langle\rangle\ Tr(\langle\rangle_{\ge d-1}\ P)) & \text{if } d>1 \\
Tr(\langle\rangle_{>d}\ P)\ = & Tr(\langle\rangle_{\ge d-1}\ P) & \\
Tr(P\ U_{\sim d}\ Q) = & Tr(\langle\rangle_{\sim d}\ Q) \wedge (P\ W\ Q) &
\end{array}$$

The translation rules assume that in the LTS model to be verified time progresses without bound and tick is not in the initiating or terminating events of any fluent. In LTSA, the fact that time progresses without bound (i.e. non-zeno) can be verified automatically by checking the following progress property requiring tick to always eventually occur:

```
progress TimeProgress = {tick}
```

The proof that the translation rules are correct involves establishing that the semantics of the bounded operators with the given temporal distance function is equivalent to the semantics of their translation in unbounded FLTL.

Intuitively, the translation of $[]_{<d}$ P when d = 1 says that P remains true unless tick and P are true. This ensures that as long as no tick event occurs P is true. The fact that P is still true when the first tick event occurs ensures that P is true if tick is the event that occurs at the position where $[]_{<1}$ P is evaluated. (It does *not* guarantee that P holds for all states that are less *or equal* to one time unit from the current position, i.e. $[]_{\le 1}$ P, because any event making P false occurring between the first and the second next ticks would make $[]_{\le 1}$ P false).

The translation rules of the $\langle\rangle_{<d}$ P operator uses the assumption that time progresses without bound. For example, the translation of $\langle\rangle_{<d}$ P when d = 1 says that from the current position, tick should not occur unless at some point in the future P holds. Since tick is assumed to always eventually occur, this guarantees that P will eventually hold and that no tick event has occurred before then. The translation of $\langle\rangle_{<d}$ P when d>1 is equivalent to the negation of the translation of $[]_{<d}$ P when d>1 (see [19] for useful equivalences of LTL formulae). This translation rule can also be understood by observing that the resulting formula is equivalent to ((! tick W P) $\vee$ (! tick W X $\langle\rangle_{<d-1}$ P)) saying that either P holds before the first occurrence of tick or $\langle\rangle_{<d-1}$ P holds just after the first occurrence of tick.

We do not explain the other translation rules, which are easier to understand.

Once translated into unbounded FLTL, assertions involving bounded operators can be model checked as any other FLTL assertions. For example, Figures 3 and 4 depict the property LTS generated by the tool for the last two assertions of Section 3.1. The LTS generated form the first property in Section 3.1 is the same as the one depicted in Figure 2.

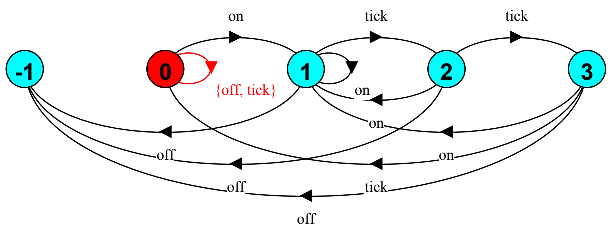

Figure 3. Property LTS for [] (on -> $[]_{<3}$ LightOn)

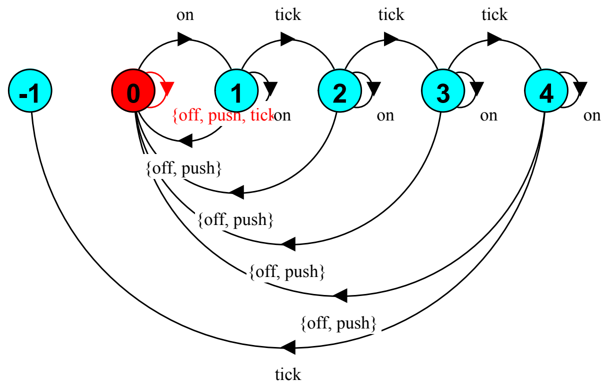

Figure 4. Property LTS for [](on -> $\langle\rangle_{\le 3}$ (off $\vee$ push))

4. EXAMPLE: THE LIGHT CONTROLLER

This section illustrates discrete-time model checking in the LTSA toolset on the light control problem [22].

4.1 Problem Statement

Consider an automated light controller used to control the lights in a room. The controller has to ensure that the lights are automatically turned on as soon as someone enters the room and the lights are automatically turned off T1 time units after the room has become unoccupied. Presence inside the room is detected by a movement detector.

The light level can be adjusted manually with a dimmer. There is a default setting that must be restored automatically if the room becomes reoccupied more than T2 time units after a movement was last detected (T2 > T1). If the room becomes reoccupied less than T2 time unit after a movement was last detected, the lights must be turned on at the level they were the last time the room was occupied.

4.2 Architecture Model

The system is composed of a light, a light controller, a movement detector. An additional process models the behaviours of users. Interfaces between these components is shown in the Figure 5.

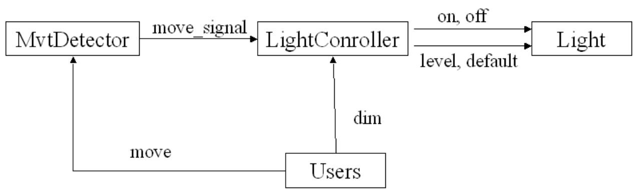

Figure 5. Architecture the Light Controller Example

The behaviour of the light controller is specified by the FSP model in Figure 6. The behaviour models of the other components are not shown. It is assumed that when a user is inside the room, a movement is detected at least once per time unit.

```
LightController = (tick -> Off_Default),
Off_Default =
    ( tick -> Off_Default
    | move_signal -> on -> On_Default[T1]
    | dim -> level -> on -> On_Dimmed[T1][T2] ),
On_Default[i:0..T1] =
    ( when (i==0) off -> Off_Default
    | when ! (i==0) tick -> On_Default[i-1]
    | when ! (i==0) move_signal -> On_Default[T1]
    | when ! (i==0) dim -> level -> On_Dimmed[T1][T2],
On_Dimmed[i:0..T1][j:0..T2] =
    ( when (i==0) off -> Off_Dimmed[j]
    | when ! (i==0) tick -> On_Dimmed[i-1][j-1]
    | when ! (i==0) move_signal -> On_Dimmed[T1][T2]
    | when ! (i==0) dim -> On_Dimmed[i][j]),
Off_Dimmed[j:0..T2] =
    ( when (j==0) default -> Off_Default
    | when !(j==0) tick -> Off_Dimmed[j-1]
    | when !(j==0) move_signal -> on  -> On_Dimmed[T1][T2]
    | when !(j==0) dim -> on -> On_Dimmed[T1][T2]).
```

Figure 6. FSP Model of the light controller

4.3 Modelling the Properties

The requirements that the light must be turned on when the room becomes occupied may be specified in discrete-time FLTL as follows:

```
assert LightTurnedOnWhenFirstEntry
        = [] (first_entry -> <>{<1} LightOn )
```

In this assertion, the fluent LightOn is defined as follows:

```
fluent LightOn = <on, off>
```

and the event first_entry is an event in the Users process denoting that a first user enters the previously unoccupied room.

The requirement that the light must remain on during T1 time units after the last exit from the room may be specified as follows:

```
assert LightOnDuringT1AfterLastExit
        = [] ( last_exit ∧ LightOn -> []{<T1} LightOn )
```

The requirement that the light must be turned off when the room has been unoccupied for more than T1 units may be specified as follows:

```
assert LightOffWhenUnoccupiedDuringT1
        = [] ( ! Occupied -> <>{<=T1} (Occupied || !LightOn) )
```

where fluent Occupied is defined as follows:

```
fluent Occupied = <first_entry, last_exit>.
```

This assertion specifies that if the room is unoccupied, then within T1 time units either the room is reoccupied or the lights are turned off.

We now consider the properties concerning light settings. A first property is that a dimmed setting has to be maintained during T2 time units after the room becomes unoccupied:

```
assert DimmedMaintainedDuringT2AfterExit
        = [] (last_exit && Dimmed -> []{<T2} Dimmed)
```

The fluent Dimmed is defined as follows:

```
fluent Dimmed = <level, default>
```

Note that the event level and default set the lights level settings but do not turn the lights on. Therefore, the lights are not necessarily on when Dimmed holds.

Conversely, after T2 time units the default light setting is restored:

```
assert DefaultRestoredWithinT2AfterExit
        = [](last_exit -> <>{<=T2} (Occupied || ! Dimmed))
```

4.4 Model Analysis

Deadlock and progress analysis allowed us to detect and correct errors in our initial version of the light controller model. For example, in the LightController process in Figure 6, an off transition from a process On_Dimmed[i][j] was leading to a process Off_Dimmed[j-1] instead Off_Dimmed[j]. The error was revealed as a deadlock. The model in Figure 6 contains no deadlock and no violation of the time progress property.

The light controller model in Figure 6 composed with environment processes satisfies the five properties in the previous section. In our initial model, we had inadvertently omitted the 'on' event in the third line of the OffDimmed process. The error was revealed by the following error trace violating the property LightTurnedOnWhenFirstEntry:

```
tick
first_entry
move_signal
on             LightOn
tick           LightOn
dim            LightOn
level          LightOn
last_exit      LightOn
move_signal    LightOn
tick           LightOn
tick           LightOn
off
first_entry
move_signal
tick
```

The column on the left shows the sequence of events that leads to the error, the column on the right shows the fluents involved in the assertion that hold after the occurrence of each event.

During the last time period, a first_entry event occurs and the light is not turned on before the following occurrence of a tick event. The model is checked with T1 = 2. Examining the error trace reveals that the error occurs after the light has been dimmed (during the first time unit) and automatically turned off (during the fourth time unit).

Model checking can also be used to generate witness system executions. For example, a witness execution satisfying the goals

LightOnDuringT1AfterLastExit can be generated by model checking the following stronger assertion with a longer time bound:

```
assert Witness_NotLightOnDuringMoreThanT1AfterLastExit
        = [](last_exit && LightOn-> []{<T1+1} LightOn )
```

As expected, this assertion is violated and the model checker generates the following system execution in which the light remains on during 2 time units (T1 = 2) after the last time the room was occupied but not longer:

```
tick
first_entry
move_signal
on               LightOn
tick             LightOn
last_exit        LightOn
move_signal      LightOn
tick             LightOn
tick             LightOn
off
```

5. SYNCHRONOUS DISCRETE-TIME FLTL

Fluent temporal logic in [5] is an asynchronous temporal logic because its properties describe sequences of system states observed after each occurrence of an event. Synchronous temporal logic is an alternative paradigm in which properties describe sequence of states observed at a fixed time rate. Both variants are used in practice.

Temporal logic operators have very different meanings in synchronous and asynchronous temporal logics. For example, 'X P' in an asynchronous temporal logic means 'P holds after the next event', whereas in a synchronous temporal logic it means 'P holds at the next time unit'. Similarly, '[] P' in an asynchronous temporal logic means 'P holds after each event' whereas in a synchronous temporal logic it means 'P holds at each time point'. Confusion between the two variants of temporal logic may lead to important errors in the formal specification of system properties.

Synchronous temporal logic is used by the KAOS goal-oriented requirements elaboration method for the formal specification of system goals and requirements. KAOS provides systematic support for the gradual identification and formal specification of system requirements through goal refinements, goal conflict analysis and obstacle analysis.

The purpose of this section is to define a synchronous variant of fluent temporal logic so that properties elaborated using goal-oriented requirements elaboration techniques can be used as assertions to be model-checked in LTSA.

In this section we first define the synchronous fluent linear temporal logic, then provide example of synchronous FLTL assertions and discuss some differences between synchronous and asynchronous FLTL. Finally, we describe how to model check synchronous FLTL assertions by translating them into asynchronous FLTL assertions and discusss the handling of events in synchronous FLTL assertions which requires special treatment.

5.1 Syntax and Semantics

The syntax of synchronous FLTL is the same as that of asynchronous FLTL; well-formed synchronous FLTL assertions are assertions formed with standard LTL operators extended with bounded temporal operators and whose atomic propositions are fluents.

The semantics of synchronous FLTL is defined by relating event-based traces to state-based traces modelling the system state at the successive occurrences of tick.

Let *tr* be an infinite sequence of events and $h_{async} = StateTrace(tr)$ be the asynchronous state-based trace associated to *tr* (see Section 2.4). There is one and only one mapping *TickPosition: Nat -> Nat* that assigns to every time point i the position of the ith occurrence of tick in *tr*, i.e. $TickPosition\ (i) = j$ iff $i = \#\{k \in 0..j \mid tr(k) = tick\}$. The synchronous state-based trace associated to *tr*, noted $h_{sync} = Sync\text{-}StateTrace(tr)$ is defined as follows:

$$h_{sync}\ (i) = h_{async}\ (TickPosition\ (i)) \quad \text{for all } i \in Nat$$

i.e. the set of fluents that hold at the ith position of the synchronous state-based trace is the set of fluents that hold at the ith occurrence of a tick event in the asynchronous state-based trace. The temporal distance function defining the semantics of bounded temporal operators is given by $dist(i,j) = |j-i|$.

A synchronous FLTL assertion *P* is then said to be satisfied by a sequence of events *tr*, noted $tr \models_{Sync\text{-}FLTL} P$, iff $Sync\text{-}StateTrace(tr) \models P$.

Synchronous FLTL is less expressive than asynchronous FLTL because satisfaction of its assertions depends on fluent values at the occurrence of tick events only. Synchronous FLTL assertions cannot constrain the occurrence of events between two occurrences of tick. While being less expressive in terms of event-based traces, some properties are expressed in a more natural way in synchronous temporal logic than in asynchronous FLTL. Differences between synchronous and asynchronous FLTL are discussed in the following section.

5.2 Differences with Asynchronous FLTL

This section presents examples of synchronous FLTL assertions and discusses differences between synchronous and asynchronous FLTL.

5.2.1 The 'Always' Temporal Operator

As mentioned before, the 'always' temporal operator has very different interpretations in the synchronous and asynchronous FLTL. This may be a source of errors if a property written in synchronous temporal logic is interpreted as a property in asynchronous temporal logic, or vice-versa.

Consider the mine pump problem [11, 10], which we discuss in more detail in the Section 6 and Figure 7, and the property requiring that when the water level is high, the pump must be on. In synchronous FLTL this property may be specified as follows:

```
[](HighWater -> PumpOn)
```

The fluents involved in this assertion are defined as follows:

```
fluent HighWater = <water[High…Max], water[0…High-1] >
fluent PumpOn = <start, stop>
```

The meaning of this assertion in synchronous FLTL is that at every time point if the water level is high (that is that the water level is between constants High and Max), the pump must be on.

If the same assertion is interpreted as an asynchronous FLTL assertion, it has a very different meaning: it requires the pump to be on when the water level is high *after the occurrence of each event*. In asynchronous FLTL, this assertion requires that the pump be on at all events satisfying HighWater. Modellers may not realize that this assertion prevents the water level from rising above High when the pump is off. This is due to the fact that in an asynchronous trace with interleaving semantics, the event start may not occur concurrently with changes in water level.

The problem does not exist for the synchronous interpretation of the assertion because in that framework, the pump and the water level can both change value within the same time unit.

5.2.2 'Next' and Closure under stuttering

Consider now the property requiring that when the water level is high, the pump must be on *at the next time unit*. In synchronous temporal logic, this property is specified as

```
[](HighWater -> X PumpOn).
```

The same assertion in FLTL does not specify the required property correctly because the X operator means 'after the next event' instead of 'at the next time point'.

In asynchronous FLTL, some assertions involving 'next' are not closed under stuttering [13]. In our event-based framework, an assertion is said to be closed under stuttering if its satisfaction is the same for event traces that differ only by unobservable τ events. The above assertion involving next is not closed under stuttering.

Assertions that are not closed under stuttering should not be used to specify system properties because their satisfaction is not preserved by refinements of the event-based model. Invariance under stuttering is also needed for the use of partial order reduction techniques that are critical to the success of any LTL model checking procedure [8].

Interestingly, all synchronous FLTL assertions are closed under stuttering because the satisfaction of a property depends only on the values of fluents at the occurrence of tick events and these values are unaffected by the occurrences of τ events.

5.2.3 Bounded Temporal Operators

The semantics of bounded temporal operators is also slightly different in synchronous and asynchronous FLTL.

The synchronous assertion $[]_{<d}$ P is weaker than the asynchronous one because it does not constrain P to be true between tick events. The synchronous version is sometimes more appropriate than the asynchronous one when one needs to model requirements in which the property P may be temporarily violated between ticks, as in the example of Section 4.2.1.

The synchronous assertion $<>_{<d}$ P is stronger than the asynchronous one because it requires P to hold at some occurrence of a tick event, while the asynchronous one requires P to hold at the occurrence of any event, even if P becomes false before the next tick event occurs. Deciding which operator to use will depend on the problem and the property to be specified.

5.3 Model-Checking Synchronous FLTL Assertions

Synchronous FLTL assertions are model-checked by translating them into untimed asynchronous FLTL. The Translation function *Tr: FLTL*$_{Sync}$ *->* *FLTL*$_{Async}$ is defined recursively as follows:

Tr([] P)	=	[] (tick -> *Tr*(P))
Tr(<> P)	=	<> (tick $\wedge$ *Tr*(P))
Tr(P U Q)	=	(tick -> *Tr*(P)) U (tick $\wedge$ *Tr*(Q))
Tr(X P)	=	X ($\neg$ tick W (tick $\wedge$ *Tr*(P)))

Boolean operators remain unchanged (i.e. *Tr*(not P) = not *Tr*(P), etc.). The translation of bounded temporal operators consists in expanding them into assertions involving the synchronous X operator as outlined in Section 2.2.

Example. The synchronous FLTL assertion

```
[](HighWater -> X PumpOn)
```

is translated into the asynchronous FLTL assertion

[](tick -> (HighWater -> X ($\neg$ tick U (tick $\wedge$ PumpOn))))

As required, the translation rules ensure that fluent values are evaluated at the occurences of tick events only. The event-based interpretations of a synchronous FLTL assertion are therefore the same as those of its translation into asynchronous FLTL.

For example, the translation of []P means that P must hold every time a tick event occurs. Similarly, the translation of <> P says that P must eventually hold when some tick occurs. The translation of the synchronous 'next' operator is slightly more complicated. It encodes the property that P must be true at the next occurrence of a tick event. This is done by saying that just after the current event (possibly a tick event) there should be no tick event until there is a tick and P holds. This ensures that if at the next occurrence of tick, P does not hold, the translation of the synchronous assertion X P is false, otherwise it is true. This translation rule is based on the assumption that time progress without bound and can therefore use the awaits temporal operator (W) instead of the until operator (U).

5.4 Handling Events in Synchronous FLTL

The previous sections do not handle the case of fluents associated to events. As explained in Section 2.4, in FLTL the implicit fluent associated to an event becomes false as soon as another event occurs. This is not adequate for synchronous FLTL because the fluent denoting the occurrence of an event should remain true until the next tick event.

We therefore need to introduce explicit fluents

```
Occurs[ev:Event]
```

denoting that event e as occurred during the last time unit.

These fluents cannot be simply defined as

```
fluent Occurs[e:Event] = <e, tick>.
```

because we need Occurs[e] to still be true when tick occurs and fluents in [5] have been defined so that they hold over intervals that are closed on the left and open on the right. This means that an event has an effect on the *current* values fluents. Therefore, contrary to what is needed, Occurs[e] would be false when tick occurs.

To solve that problem we need to introduce a toc event that always occurs just after a tick event and define the fluent associated to event as follows:

```
fluent Occurs[e] = <ev, toc>.
```

This ensures that Occurs[e] is still true at the occurrence of tick.

The FSP process that ensures that every tick event is immediately followed by a toc event is given by

```
TickToc = (tick -> toc -> TickToc | {AllEvents} -> TickToc).
```

In this process, AllEvents is the alphabet of the LTS model being verified. Referring to the complete alphabet of the LTS model guarantees that tick is *immediately* followed by a toc with no other events in between. This is needed because if during some time unit an event e could occur between the tick and the toc, Occurs[e] will be false at the next occurrence of tick.

Note that, contrary to [5], the fluents of Miller and Shanahan [21] hold over intervals that are *open* on the left and *closed* on the right. This means that an event has an effect on the values of fluents in the *next* state. In this framework, tick could be defined as terminating event for Occurs[ev] and we wouldn't need toc events. The translation rules of Section 4.2 would remain the same in that framework because, since domain fluents (i.e. fluents that are not associated to the occurrence of an event) do not have tick among their initiating or terminating events, they have the same value at the occurrence of a tick (as evaluated in [5]) and just after (as evaluated in [21]).

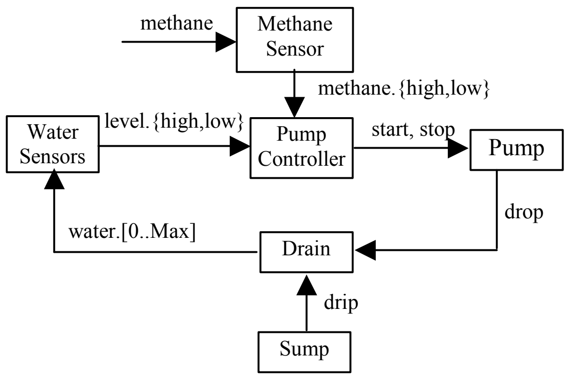

Figure 7. Architecture of the Mine Pump Case Study

6. EXAMPLE: THE MINE PUMP

This section illustrates model checking of synchronous FLTL assertions on the mine pump case study [11, 10].

A FSP model for the mine pump system had been developed previously to illustrate the SceneBean animation feature of the LTSA toolset [18]. The structure of this model is shown in the Figure 7.

The properties to be satisfied by the mine pump control system may be modelled in synchronous FLTL as follows:

```
assert PumpOnWhenHighWaterAndNoMethane
                    = [](HighWater && ! Methane -> PumpOn)
assert PumpOffWhenLowWater = [](LowWater -> ! PumpOn)
assert PumpOffWhenMethane = [](Methane -> ! PumpOn)
```

These properties correspond to well known properties of the mine pump problem as they are frequently specified in requirements models [10]. The first property requires the pump to be on when the water level is high and there is no methane, the second requires the pump to be off when the water level is low and the third on requires the pump to be off when there is methane.

For more complex systems in which the identification and formalisation of the required properties might be more difficult than for the mine pump, a goal-oriented method such as KAOS would provide guidance for identifying and specifying the properties to be satisfied by the system. A KAOS goal model of the mine pump system containing the above properties may be found in [15].

In order to interpret KAOS goals as synchronous FLTL assertions, currently one also needs to manually provide the fluent definitions that relate the predicates involved in the goal definitions to the events appearing in the behaviour model. In this case, the fluents appearing in the required properties are defined as follows:

```
fluent HighWater = <water[High..Max], water[0..High-1]>
fluent LowWater = <water[0..Low], water[Low+1..Max]>
                                        initially True
fluent Methane = <methane.high, methane.low>
fluent PumpOn = <start, stop>
```

This step of relating fluents to events may be delicate. In the future, we intend to provide automated support for deriving fluent definitions from a KAOS operational model.

In our FSP model of the mine pump system, there are two different models for the mine pump controller. In a first model, the controller is unsafe because it ignores the presence of methane. Checking the model against the above properties generates a counter-example in which the water level raises to the high position (water.11 in our model), the pump is turned on, methane appears and the pump is not turned off before the following tick. The last two time units of the generated counter-example are shown below:

```
…
tick
water.10
tick
drip
water.11
level.high
start           PumpOn
drop            PumpOn
methane         PumpOn
methane.high    Methane && PumpOn
tick            Methane && PumpOn
```

The second model for the pump controller resolves the problem by ensuring that the pump is turned off when methane is detected. This new model satisfies all three properties.

Note that, as discussed in Section 5.2, the same assertions interpreted as asynchronous FLTL assertions would not correctly capture the intended system properties. In asynchronous FLTL, these assertions are in fact too strong and are violated by the correct specification of the pump controller.

7. CONCLUSION

Fluent linear temporal logic is a convenient formalism for modelling and reasoning about properties of event-based systems such as those used to describe software architectures.

This paper has presented two extensions of FLTL for modelling properties of discrete-time event-based models.

Firstly, we have extended FLTL with bounded temporal logic operators allowing modellers to easily specify timed properties of discrete-time event-based models and have defined an encoding of these new operators into untimed FLTL so that they can be model-checked into the untimed framework.

Secondly, we have considered a synchronous variant of fluent temporal logic for describing sequence of states observed at a fixed time rate, rather than after each occurrence of an event. The synchronous approach is a natural way to model properties of discrete-time *state-based* models. It is used in particular in the KAOS goal-oriented requirements elaboration method for the formal specification of goals and requirements. We have seen that the interpretation of temporal logic operators is different in synchronous and asynchronous temporal logics with important risks of confusion between the two. Our aim in this paper was to clarify the differences between the two existing paradigms and to define a mapping from synchronous FLTL to asynchronous FLTL so that properties written in synchronous FLTL can be model checked in the existing asynchronous framework.

It is not yet clear to us whether the two flavours of temporal logic specifications need to be kept in the long term or not.

The work reported in this paper is part of our larger effort to integrate goal-oriented requirements elaboration methods with automated techniques for the formal analysis of event-based models such as those implemented in the LTSA toolset. A recurrent problem in the use of model-checking tools concerns the difficulty of identifying and correctly specifying the required system properties [4, 9]. This paper partly address this problem by allowing LTSA modellers to use a goal-oriented requirements elaboration process à la KAOS for the incremental identification, elaboration and specification of the formal properties to be model-checked with the LTSA toolset. Future work will define how to automatically derive an event-based LTS model from a KAOS operation model so that KAOS modellers can use the LTSA toolset for the formal analysis of their operational models. We also wish to explore a more constructive process for elaborating event-based models of software architectures from declarative specification of requirements expressed in fluent temporal logic.

ACKNOWLEDGEMENT
The work reported herein was partially supported by the Belgian "Fond National de la Recherche Scientifique" (FNRS) and EPSRC grant READS GR/S03270.

8. REFERENCES

[1] R. Allen and D. Garlan, A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology (TOSEM),* Vol. 6, No. 3, pp. 213-249, 97.

[2] M. Bernardo, P. Ciancarini and L. Donatiello, Architecting Software Systems with Process Algebras, University of Bologna, UBLCS-2201-7, July 2001.

[3] R. Cleaveland, J. Parrow and B. Steffen, The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems, *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 1, pp. 36-72, 1993.

[4] M.B. Dwyer, G. S. Avrunin and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification", *Proc. ICSE'99 - 21st Intl. Conference on Software Engineering*, Los Angeles, May 1999.

[5] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-Based Systems", in Proc. of the 4th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), September 2003, Helsinki, Finland.

[6] T. A. Henzinger. It's about time: Real-time logics reviewed. *Proc. 9th International Conference on Concurrency Theory (CONCUR),* LNCS 1466, Springer, 1998, pp. 439-454.

[7] C. A. R. Hoare, *Communicating sequential processes*, Prentice-Hall International, 1985.

[8] G.J. Holzmann, The Model Checker Spin, *IEEE Transactions. on Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.

[9] G.J. Holzmann, The Logic of Bugs, *Proc. ACM Foundations of Software Engineering (FSE)*, Charleston SC USA, November 2002.

[10] M. Joseph, *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall, 1996.

[11] J. Kramer, J. Magee, M. Sloman et al, CONIC: an Integrated Approach to Distributed Computer Control Systems. *IEE Proceedings*, Part E 130, 1, January 1983.

[12] R. Koymans, Specifying message passing and time-critical systems with temporal logic, LNCS 651, Springer, 1992.

[13] L. Lamport, The Temporal Logic of Actions, *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, pp. 872-923, 1994.

[14] A. van Lamsweerde, Requirements Engineering in the Year 00: A Research Perspective, *22nd International Conference on Software Engineering*, Limerick, ACM Press, 2000.

[15] E. Letier, Reasoning about Agents in Goal-Oriented Requirements Engineering. Ph. D. Thesis, University of Louvain, May 2001.

[16] J. Magee, N. Dulay, S. Eisenbach and J. Kramer, Specifying Distributed Software Architectures, 5th European Software Engineering Conference (ESEC'95), Sitges, Spain, 989, pp. 137-153, September 1995.

[17] J. Magee and J. Kramer, Concurrency - State Models & Java Programs, Chichester, John Wiley & Sons, 1999.

[18] J Magee, N Pryce, D Giannakopoulou, J Kramer, Graphical animation of behavior models, 22nd International Conference on Software Engineering, Limerick, 2000.

[19] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag, 1992.

[20] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[21] R. Miller and M. Shanahan, *The Event Calculus in Classical Logic - Alternative Axiomatisations*, Linkoping Electronic Articles in Computer and Information Science, Vol. 4, No. 16, pp. 1-27, 1999.

[22] S. Queins et al., The Light Control Case Study: Problem Description, *Journal of Universal Computer Science*, Special Issue on Requirements Engineering: the Light Control Case Study, Vol.6(7), 2000.

[23] A. W. Roscoe, A Classical Mind: Essays in Honour of C.A.R. Hoare, pp. 353-378, Prentice-Hall, 1994.