Experimenting a temporal logic for executable specifications in an engineering domain

L. Chittaro & A. Montanari
Dipartimento di Matematica e Informatica, Università di Udine, Via Zanon, 6, 33100 Udine, Italy

ABSTRACT

This paper reports the results of an experiment with a logical specification language on an engineering case study. The proposed language extends the Event Calculus formalism with primitives for modeling context-dependency and discrete processes. In particular, it introduces a number of process constructors that provide a way of packaging related events into individual, conceptual chunks. It has been implemented on a Sun SPARC2 with Quintus Prolog and used to write executable specifications of a gas heater system. The paper discusses in detail the problems encountered in executing the calculus with Prolog.

1 INTRODUCTION

This paper reports the results of an experiment with a logical specification language on an engineering case study. We define a language that extends the Event Calculus [13, 7, 20, 16] with primitives for modeling context-dependency and discrete processes. Our extensions are a first step in the development of a formalism able to represent and reason about the temporal behaviour of complex systems. It is part of a long-term research activity devoted to the development of a uniform logical calculus of discrete, continuous and abnormal processes. Our concern goes also to efficiency issues which can not be avoided in any realistic application.

A great deal of work concerning the definition of formal languages to specify concurrent and real-time systems was carried out by the software engineering community. At the same time, the development of languages for the representation of temporal knowledge has always been central in several areas of artificial intelligence, e.g. natural language processing and planning. Last year saw a growing interest in temporal reasoning in the
Artificial Intelligence in Engineering

emerging field of model-based reasoning about physical and engineering systems, where both the need for a temporal model in representing behavioural knowledge (e.g. [5]) and the need for temporal abstraction mechanisms (e.g. [9]) have been stressed.

In principle, our calculus can be embedded in any system requiring temporal reasoning capabilities, e.g. systems for envisionment, planners, diagnostic systems, and so on. In this paper we show how the proposed formalism can be used to write executable specifications of a gas heater. The proposed calculus allows one to model the desired behavior of each system process taken in isolation (in the example, the process of boot up, heating cycle, shut down, etc.) and to properly constrain the composition of processes. The adequacy of the resulting specification can then be tested by providing sets of events describing admissible evolutions of the system, called system histories, and proving that they are compatible with the specification.

The paper is organized in the following way. In the next section, we illustrate the gas heater system case study. Then, in section 3 we present the logical calculus of discrete processes. Section 4 discusses some excerpts from the specification of the gas heater behaviour and presents some execution examples. Last section is devoted to the analysis of looping problems and computational complexity and identifies the major requirements for an efficient implementation of the calculus.

A CASE STUDY: THE GAS HEATER

Consider the following representational problem. We want to represent a gas heater, as shown in Figures 1 and 2. Various events can occur and we want to be able to predict their consequences.

A user perspective

The gas heater presents to its user an interface (shown in Figure 1) composed by two buttons (Safety Disable and Lighter) which must be used during the system start up procedure, a switch (Power) which is used to enable or disable system operation, a knob (Desired Temperature) which is used to select the desired temperature for the environment where the gas heater is placed, a tap (Gas Main) which is used to allow or prevent the supply of gas to the heater and a plug to supply electrical power to the heater.

The user typically starts up the system by: connecting the plug to a socket, opening the Gas Main tap, turning on the Power switch, pressing the Safety Disable button together with the Lighter button and keeping them pressed until he/she sees the pilot light turning on, releasing then the two buttons. When the pilot light is on, heating is automatically controlled by the system, which burns gas when room temperature is less than desired and keeps only the pilot light on when room temperature is equal to or
greater than desired.

An engineer perspective

Looking inside the gas heater, six main components (shown in Figure 2) deserve attention in order to understand how the gas heater accomplishes its purpose. The Power Switch allows or prevents supply of electrical power to the Thermostat and to the Lighter System.

The Lighter System is devoted to produce sparks in order to light up the pilot light during the start up procedure of the gas heater. The Safety System is devoted to prevent dangerous gas leaks into the environment: it is a thermo-mechanical system (so it can operate also during blackouts) which closes a valve (Safety Valve) when it is not heated by the pilot light and opens the same valve when the pilot light is on. If the Safety Disable
button is pressed, the Safety Valve allows gas flow through the pipe connected to the pilot light (thus allowing, if needed, to ignite the pilot light with the Lighter) but not through the pipe connected to the Thermostatic Valve. The Thermostat senses room temperature and opens the Thermostatic Valve if the temperature is lower than the Desired Temperature set by the user; it closes the valve when temperature is equal to or higher than the Desired Temperature set by the user. When both the Safety Valve and the Thermostatic Valve are open, a huge quantity of gas is allowed to reach the pilot light and be burnt.

A LOGICAL CALCULUS OF DISCRETE PROCESSES

This paper presents a calculus of processes based on the model of time and change of the Event Calculus. The Event Calculus proposes a general approach to represent and reason about events and their effects in a logical framework [8]. From a description of events that occur in the real world, it allows one to derive various relationships and the time periods for which they hold. It also embodies a notion of default persistence, that is, relationships are assumed to persist until an event occurs which terminates them. As an example, if we know that an aircraft enters a given sector at 10:00hrs and leaves at 10:20hrs, the Event Calculus allows us to infer that it is in that sector at 10:15hrs. More precisely, the Event Calculus takes the notions of event, property, time-point and time-interval as primitives and defines a model of change in which events happen at time-points and initiate and/or terminate time-intervals over which some property holds. So, for instance, the events of entering and leaving the sector initiate and terminate the aircraft’s property of being in the sector, respectively. Time-points are unique points in time at which events take place instantaneously. In the previous example, the event of entering the sector occurs at 10:00hrs, while the event of leaving the sector occurs at 10:20hrs. Time-intervals are represented by means of tuples of two time-points. With the same example, we can deduce that the aircraft is in the sector during the time-interval starting at 10hrs and ending at 10:20hrs. A specific domain evolution is modeled by a set of event occurrences. Each event occurrence is obtained by attaching a time-point to the event type. The pair (event, time-point) univocally identifies an event occurrence provided that two events of the same type can not simultaneously happen. If this assumption is no longer valid, explicit identifiers for event occurrences have to be added. Formally, we represent event occurrences by means of the happens_at predicate: happens_at(event, Time).

Moreover, we represent (part of the) domain knowledge by means of initiates_at and terminates_at predicates that express the effects of events on properties:

\[
\text{initiates}_\text{at}(E, Q, T) \leftarrow \quad \text{terminates}_\text{at}(E, Q, T) \leftarrow \\
\text{happens}_\text{at}(E, T) \quad \quad \text{happens}_\text{at}(E, T)
\]
The relations initiates_at and terminates_at state that each instance of event initiates and terminates a period of time during which property holds, respectively. Both of them are context-independent, that is, the occurrence of an event of the given type initiates, or terminates, the validity of the relevant property whatever is the context in which it occurs.

The basic calculus of events

The basic model of time and change is defined by means of a set of basic axioms. The first axiom we introduce is the mholds_for. It allows us to state that the property P holds maximally (i.e. there is no larger time-interval for which it also holds) over \([\text{Start}, \text{End}]\) if an event \(E\) occurs at the time \(\text{Start}\) which initiates P, and an event \(E'\) occurs at time \(\text{End}\) which terminates P, provided there is no known interruption in between:

\[
\text{mholds_for}(P, [\text{Start}, \text{End}]) \leftarrow \text{broken_during}(P, [\text{Start}, \text{End}]) +
\text{initiates_at}(E, P, \text{Start}) \land \text{terminates_at}(E, P, \text{T}) \land \\
\text{terminates_at}(E', P, \text{End}) \land \text{Start} < \text{T} \land \text{End} > \text{T} \land \\
\text{End} > \text{Start} \land \\
\text{not broken_during}(P, [\text{Start}, \text{End}])
\]

The negation involving the broken_during predicate is interpreted using negation-as-failure. This means that properties are assumed to hold un-interrupted over an interval of time on the basis of failure to determine an interrupting event. Should we later record a terminating event within this interval, we can no longer conclude that the property holds over the interval. This gives us the non-monotonic character of the calculus which deals with default persistence. Formally, the broken_during predicate states that a given property \(P\) ceases to hold at some point during the time-interval \([\text{Start}, \text{End}]\) if there is an event which terminates \(P\) at a time \(T\) within \([\text{Start}, \text{End}]\).

We also define the iholds_for predicate in terms of mholds_for to state that a property holds over each time-interval included in the maximal one and the holds_at predicate which is similar to iholds_for except that it relates a property to a time-point rather than to a time-interval:

\[
iholds_for(p, [\text{Start}, \text{End}]) \leftarrow \text{holds_at}(P, T) \leftarrow \\
\text{mholds_for}(P, [\text{A}, \text{B}]) \land \text{Start} \geq A \land \text{End} \leq B \\
\text{mholds_for}(P, [\text{Start}, \text{End}]) \land \\
T > \text{Start} \land T \leq \text{End}
\]

In particular, the holds-at predicate conventionally states that a property is not valid at the left endpoint of the maximal time-interval, while it is valid at the right one.

Context dependency

To make initiates_at and terminates_at relations context-dependent, we must be able to state that the occurrence of an event of the given type at a certain
time point initiates, or terminates, the validity of the associated property provided that some preconditions hold at such an instant. Formally, context-dependent initiates_at and terminates_at relations are defined as follows:

\[
\text{initiates_at}(E, Q, T) \leftarrow \begin{align*}
\text{happens_at}(E, T) \land \\
\text{holds_at}(\text{Prec1}, T) \land \\
\ldots \land \\
\text{holds_at}(\text{PrecN}, T)
\end{align*}
\]

\[
\text{terminates_at}(E, Q, T) \leftarrow \begin{align*}
\text{happens_at}(E, T) \land \\
\text{holds_at}(\text{Prec1}, T) \land \\
\ldots \land \\
\text{holds_at}(\text{PrecN}, T)
\end{align*}
\]

where \( N \) is greater than 0 (if it was equal to 0 then they would be context-independent).

The initiates_at and terminates_at predicates state a sort of cause-effect relationship between events and properties. They provide a plausible model of typical interactions between temporal objects in a static framework, but give no account of dynamic contexts in which entering a given state forces the occurrence of specific events.

To model these situations, we introduce the autoinitiates_at and autoterminates_at predicates relating the modification of properties to the achievement of certain domain/system configurations.

They are used to represent (part of) domain knowledge by means of the following axioms:

\[
\text{initiates_at}(\text{begin}(Q), Q, T) \leftarrow \text{happens_at}(\text{begin}(Q), T) \leftarrow
\begin{align*}
\text{happens_at}(\text{begin}(Q), T) \\
\text{autoinitiates_at}(Q, T)
\end{align*}
\]

where \( \text{begin}(Q) \) denotes an event initiating a period of time over which property \( Q \) holds, whose occurrence is caused by the configuration of the system, and \( \text{autoinitiates_at}(Q, T) \) is defined as

\[
\text{autoinitiates_at}(Q, T) \leftarrow
\begin{align*}
\text{mholds_for}(P1, [I1]) \land \ldots \land \text{mholds_for}(PN, [IN]) \land \\
\text{begin_intersection}([I1, \ldots, IN], T)
\end{align*}
\]

where the \( \text{begin_intersection} \) predicate returns the first endpoint of the intersection of the intervals \( I1, I2, \ldots, IN \) (if it exists).

The definition of the autoterminates_at\((Q, T)\) predicate is analogous:

\[
\text{terminates_at}(\text{end}(Q), Q, T) \leftarrow \text{happens_at}(\text{end}(Q), T) \leftarrow
\begin{align*}
\text{happens_at}(\text{end}(Q), T) \\
\text{autoterminates_at}(Q, T)
\end{align*}
\]

where \( \text{end}(Q) \) denotes an event terminating a period of time over which property \( Q \) holds, whose occurrence is caused by the configuration of the system, and the autoterminates_at\((Q, T)\) definition is identical to the definition of autoinitiates_at\((Q, T)\).
Discrete processes
The basic calculus deals with single and instantaneous events only. Such an assumption makes the formalism quite simple, but it is very restrictive. It allows us to use the standard ordering relations between numbers $<, >, \leq, \geq$ to compare the occurrence times of events in order to determine the periods of time over which properties hold, but it is acceptable only if the effect of the occurrence of a set of events is equal to the union of the effects of the occurrence of each single event. In general, this is not the case: the effects of a set of events occurring in a given temporal ordering may differ from the collections of the effects of the occurrence of each single event taken in isolation. We define a way of packaging related events into individual, conceptual chunks called processes. Formally, we define a number of operators, called process constructors, that allow us to specify the structure of processes by constraining the temporal relations among their components. In particular, they allow us to model basic forms of process interaction such as temporal delays between processes, sequential, simultaneous or alternative occurrences of processes, and iteration of processes [11,3]. The set of process constructors of the calculus is similar to the set of operators of path expressions. Originally developed for modeling operating system behaviour, path expressions have been successfully used in several areas of computer science, e.g. [2]. In this paper we show how they can be usefully employed to model domain and physical system behaviour. With respect to the classical works on the formalization of process interactions, e.g. [14,10,4], our research is characterized by a logical bias. We aim at developing a calculus that gives a logical meaning to operators which are operational in nature.

The set of process constructors is the following:

- **seq** sequence (binary operator)
- **delay(min, max)** minimum and maximum delay between two processes (binary operator)
- **salt** strong alternative, i.e., one and only (n-ary operator)
- **walt** weak alternative, i.e., at least one (n-ary operator)
- **par** parallelism (binary operator)
- **pseq(N)** + sequential iteration, i.e., $n$ occurrences of a given process, with $n \geq 1$ (binary operator)
- **aseq(N)** * sequential iteration, i.e., $n$ occurrences of a given process, with $n \geq 0$ (binary operator)
- **[]** composition

Formally, a process is defined as follows:

1. each atomic event is a process;
2. if $P_1, P_2, ..., P_n$ are processes, $N_1$ is a positive integer, and $Min, Max$, and $N_2$ are integer greater than or equal to 0, then $seq(P_1, P_2)$,
pseq(Pl,N1), aseq(Pl,N2), delay(Pl,P2,[Min,Max]), salt(Pl,...,Pn),
wa!t(Pl,...,Pn), and par(Pl,P2), are processes;

3. nothing else is a process.

Elementary processes are atomic events occurring at time points, structured processes are macro-events consisting of a number of subprocesses and taking place over finite time-intervals. As in the case of properties, time-intervals are represented by means of tuples of two time points [T1,T2], with T1 < T2. Furthermore, we conventionally extend this definition to pairs [T1,T2], with T1 = T2, corresponding to degenerate intervals over which atomic events occur. The meaning of process constructors is given by a suitable set of axioms. Some of them make implicit use of the usual relationships between time-intervals to constrain the occurrence times of components. These relations can be easily expressed in terms of relationships between interval endpoints as shown in [12].

Let us now define the axioms of the calculus of discrete processes:

\[
happens\_over(E,[T,T]) \leftarrow \happens\_at(E,T)
\]
\[
happens\_over(seq(Pl,P2),[T1,T4]) \leftarrow \happens\_over(Pl,[T1,T2]) \land \happens\_over(P2,[T3,T4]) \land T2 < T3
\]
\[
happens\_over(delay(Pl,P2,[Min,Max]),[T1,T4]) \leftarrow \happens\_over(Pl,[T1,T2]) \land \happens\_over(P2,[T3,T4]) \land 0 \leq Min \land Min \leq Max \land T3 - T2 \geq Min \land T3 - T2 \leq Max
\]
\[
happens\_over(salt(Proclist),[T1,T2]) \leftarrow member(Pl,Proclist) \land \happens\_over(Pl,[T1,T2]) \land not (member(P2,Proclist) \land P2 \neq P1 \land \happens\_during(P2,[T1,T2])
\]
\[
happens\_during(Pl,[T1,T2]) \leftarrow \happens\_over(Pl,[T3,T4]) \land T1 < T3 \land T4 < T2
\]
\[
happens\_over(walt(Proclist),[T1,T2]) \leftarrow member(Pl,Proclist) \land \happens\_over(Pl,[T1,T2])
\]
\[
happens\_over(par(Pl,P2),[T1,T2]) \leftarrow \happens\_at(Pl,[T1,T2]) \land \happens\_at(P2,[T1,T2]) \land P1 \neq P2
\]
\[
happens\_over(pseq(P,1),[T1,T2]) \leftarrow \happens\_over(P,[T1,T2])
\]
\[
happens\_over(pseq(P,N),[T1,T4]) \leftarrow \happens\_over(P,[T1,T2] \land \happens\_over(P,[T3,T4] \land T2 < T3 \land \happens\_over(pseq(P,N1),[T3,T4]) \land Nis \_N1 + 1
\]
Artificial Intelligence in Engineering

\[
\text{happens\_over}(\text{aseq}(P,0), [T_1, T_2]) \leftarrow \\
\text{not happens\_over}(P, [T_1, T_2])
\]

\[
\text{happens\_over}(\text{aseq}(P,N), [T_1, T_2]) \leftarrow \\
\text{happens\_over}(\text{nseq}(P,N), [T_1, T_2])
\]

With regard to the delay axiom, if \(Min = Max = 0\) a particular form of process synchronization, namely sequential synchronization, is modeled. It constrains the last endpoint of the time-interval over which the first process takes place to coincide with the first endpoint of the time-interval associated with the second process. In such a case, the axiom can be rewritten as follows:

\[
\text{happens\_over}(\text{mseq}(P_1, P_2), [T_1, T_2]) \leftarrow \\
\text{happens\_over}(P_1, [T_1, T_2]) \land \text{happens\_over}(P_2, [T_2, T_3])
\]

where \text{mseq} stands for meeting sequence.

In general, domain axioms include definition of processes in terms of a suitable composition of subprocesses. An example of these domain axioms is:

\[
\text{occurs\_process}(p, [T_1, T_2]) \leftarrow \\
\text{happens\_over}(\text{seq}(p_1, \text{salt}(p_2, \text{par}(p_3, p_4))), [T_1, T_2]) \\
\text{iholds}(\text{Prec}_1, [T_1, T_2]) \land ... \land \text{iholds}(\text{Prec}_N, [T_1, T_2])
\]

where \(p\) is the (type of the) compound process, \(p_1, p_2, p_3,\) and \(p_4\) are the (types of the) component processes, and \(\text{Prec}_1, ..., \text{Prec}_N\), with \(N\) greater than or equal to 0, are properties that must be valid in each instant of the time-interval over which the process takes place. Properties \(\text{Prec}_1, ..., \text{Prec}_N\) express conditions that have to be preserved during the process occurrence to guarantee process integrity. Their role is similar to the role of protections in planning systems [19,15].

Any structured component is substituted with its own definition by means of the axiom:

\[
\text{happens\_over}(\text{Pname}, [T_1, T_2]) \leftarrow \\
\text{occurs\_process}(\text{Pname}, [T_1, T_2])
\]

THE EXECUTABLE SPECIFICATION OF THE GAS HEATER

The specification of events and properties

The first step taken in order to represent the gas heater was the identification of the types of events which can be contained in the database. Twelve types were identified: nine of them represent the possible actions of the user of the gas heater, i.e. open or close the Gas Main tap (openGasMain and closeGasMain events), turn on or off the Power switch (switchOnPower
and switchOffPower events), press or release the Safety Disable button (pressDisableButton and releaseDisableButton events), press or release the Lighter button (pressLighterButton and releaseLighterButton events), setting the thermostat to the desired temperature (setThermostat(X) events).

Another type of event (i.e. thermometerReport(X)) represents temperature changes in the environment as reported by a thermometer (it can be assumed that this reports are directly provided by the digital thermometer that can be found in modern thermostats). The last two types of events (pilotLightIgnition and pilotLightExtinction) concern the observation of the pilot light turning on or off; the explicit introduction of this two types of events is not strictly necessary; we introduced them in this example in order to break a physical feedback loop in the gas heater model, thus avoiding the need of using a loop checker mechanism in the program (this issue is further discussed in section 5).

A possible database built using the above described types of events is for example the following:

\[
\begin{align*}
\text{happens_at(setThermostat(19),0).} \\
\text{happens_at(thermometerReport(19),1).} \\
\text{happens_at(openGasMain,1).} \\
\text{happens_at(switchOnPower,2).} \\
\text{happens_at(pressDisableButton,3).} \\
\text{happens_at(pressLighterButton,3).} \\
\text{happens_at(pilotLightIgnition,4).} \\
\text{happens_at(releaseLighterButton,5).} \\
\text{happens_at(releaseDisableButton,6).} \\
\text{happens_at(thermometerReport(18),8).} \\
\text{happens_at(thermometerReport(19),10).} \\
\text{happens_at(thermometerReport(18),11).} \\
\text{happens_at(pressDisableButton,19).}
\end{align*}
\]

In this scenario, the user initially sets the desired temperature to 19 Celsius degrees, then he/she takes all the actions needed in order to ignite the pilot light: the pilot light turns on at time 4 and the user releases the Safety Disable button at time 6; between time 8 and 11 the thermometer reports some temperature changes in the environment; at time 19 the user presses again the Safety Disable button.

The second step taken in order to represent the gas heater was the identification of the properties that can be initiated and terminated by events. These properties fall in three classes: properties initiated or terminated by the simple occurrence of an event; properties initiated or terminated by the occurrence of an event in a specific context; properties initiated or terminated by an event which is not explicitly stated in the database.

An example of the first class of properties is given by the gasSupplied property, which represents the availability of gas at the input of the Safety Valve and is initiated and terminated respectively by the openGasMain and closeGasMain events. The representation is the following:
Artificial Intelligence in Engineering

\[\text{initiates}_\text{at}(\text{openGasMain}, \text{gasSupplied}, T) :- \]
\[\text{happens}_\text{at}(\text{openGasMain}, T).\]

\[\text{terminates}_\text{at}(\text{closeGasMain}, \text{gasSupplied}, T) :- \]
\[\text{happens}_\text{at}(\text{closeGasMain}, T).\]

Another example is given by the desiredTemperature(X) property which states that X is the desired temperature for the environment. In this case, setThermostat(X) is both the initiating and terminating event: it initiates the desiredTemperature(X) property and terminates any previous desiredTemperature(Y). The representation is the following:

\[\text{initiates}_\text{at}(\text{setThermostat}(X), \text{desiredTemperature}(X), T) :- \]
\[\text{happens}_\text{at}(\text{setThermostat}(X), T).\]

\[\text{terminates}_\text{at}(\text{setThermostat}(X), \text{desiredTemperature}(Y), T) :- \]
\[\text{happens}_\text{at}(\text{setThermostat}(X), T).\]

The specification of context dependency

An example of the second class of properties is given by the sparkingLighter property, which states that the lighter is producing sparks. In the following, we represent the relations between the sparkingLighter property and the events which initiates or terminates it (the Lighter produces sparks only if the Lighter button is pressed and electrical power is provided to the lighter):

\[\text{initiates}_\text{at}(\text{pressLighterButton}, \text{sparkingLighter}, !) :- \]
\[\text{happens}_\text{at}(\text{pressLighterButton}, !), \text{holds}_\text{at}(\text{powerSupplied}, !).\]

\[\text{initiates}_\text{at}(\text{switchOnPower}, \text{sparkingLighter}, !) :- \]
\[\text{happens}_\text{at}(\text{switchOnPower}, !), \text{holds}_\text{at}(\text{pressedLighterButton}, !).\]

\[\text{terminates}_\text{at}(\text{releaseLighterButton}, \text{sparkingLighter}, !) :- \]
\[\text{happens}_\text{at}(\text{releaseLighterButton}, !).\]

\[\text{terminates}_\text{at}(\text{switchOffPower}, \text{sparkingLighter}, !) :- \]
\[\text{happens}_\text{at}(\text{switchOffPower}, !).\]

A more complex example is given by the thermostaticValveOpen property, which states that the Thermostatic Valve is currently open. In the following, we represent the relation between the thermometerReport(X) event and the thermostaticValveOpen property (the Valve opens only if the new temperature is greater than the currently desired one and if there is electrical power available to open it):

\[\text{initiates}_\text{at}(\text{thermometerReport}(X), \text{thermostaticValveOpen}, T) :- \]
\[\text{happens}_\text{at}(\text{thermometerReport}(X), T), \text{holds}_\text{at}(\text{desiredTemperature}(Y), T), X < Y, \text{holds}_\text{at}(\text{powerSupplied}, T).\]
 Artificial Intelligence in Engineering

\[
\text{terminates} \_\text{at}(\text{thermometerReport}(X), \text{thermostaticValveOpen}, T) : - \\
\text{happens} \_\text{at}(\text{thermometerReport}(X), T), \\
\text{holds} \_\text{at}(\text{desiredTemperature}(Y), T), X \geq Y.
\]

An example of the third class of properties is given by the burning property, which states that the gas heater is currently heating the environment. In the following, we represent the autoinitiates relation between the thermostaticValveOpen and the burning property (burning initiates at the time when the Thermostatic Valve is open, the pilot light is on and the Safety Valve is open):

\[
\text{autoinitiates} \_\text{at}(\text{burning}, T) : - \\
\text{mholds} \_\text{for}(\text{thermostaticValveOpen}, I1), \text{mholds} \_\text{for}(\text{pilotLightOn}, I2), \\
\text{mholds} \_\text{for}(\text{safetyValveOpen}, I3), \\
\text{begin} \_\text{intersection} \_\text{at}([I1, I2, I3], T).
\]

By using the full set of initiating and terminating predicates representing the gas heater, the PROLOG interpreter is able to answer to queries such as:

\[
?- \text{mholds} \_\text{for}(\text{burning}, I).
\]

In the case of the database presented in the previous paragraph, this query returns two solutions:

- \(I=[8,10]\); burning initiates at 8, because the temperature has dropped to 18 degrees and terminates at 10, because it has risen to 19;
- \(I=[11,19]\); burning initiates at 11, because the temperature has again dropped to 18 degrees and terminates at 19, because the user has pressed the Safety Disable button which allows gas flow only to the pilot light and not to the Thermostatic Valve.

It is also possible to pose totally unbound \text{mholds} \_\text{for} queries, obtaining as solutions all intervals of validity for each property, i.e. a complete temporal portrait of the gas heater behavior.

Even if it has been extended with primitives for context-dependency, auto-initiation and auto-termination, however, the basic calculus is still unsatisfactory, because it provides no facilities for process modeling. It gives an incomplete description of system behavior that only captures the effects of the occurrence of atomic events in a restricted number of local contexts. The next section shows how the process constructors can be successfully applied to the gas heater problem.

The specification of discrete processes

Let us report now the model of the \text{heatingCycle} structured process, that captures the gas heater normal behaviour whose purpose is keeping the
temperature room constant. At the first level of refinement, the *heating-Cycle* process is decomposed into a meeting sequence of three subprocesses: *startHeatingCycle*, *heatingSequence*, and *endHeatingCycle*.

The *startHeatingCycle* process represents the two possible ways of starting an heating cycle at an higher level of detail: the first one (*instantStart*) represents the situation where burning in the gas heater starts immediately, because the temperature is under the selected threshold; the second one (*delayedStart*) represents the situation where burning in the gas heater is delayed until the environment temperature falls under the selected threshold.

The *heatingSequence* process describes the sequence of heating processes, each one occurring when the environment temperature is to be increased.

Finally, the *endHeatingCycle* process represents the two possible ways of ending an heating cycle at an higher level of detail: the first one (*instantEnd*) represents the situation where the heating cycle ends together with the last heating process; the second one (*delayedEnd*) represents the situation where the last heating process ends before the conditions for normal operation of the gas heater fail to hold. The *delayedStart*, *heatingSequence*, and *delayedEnd* definitions also provide an example of the use of conditions to guarantee process integrity.

\[
\text{occurs\_process(heatingCycle,I):-}
\text{happens\_over(mseq(startHeatingCycle,}
\text{\quad mseq(heatingSequence,endHeatingCycle)),I).}
\]

\[
\text{occurs\_process(startHeatingCycle,I):-}
\text{happens\_over(salt([instantStart,delayedStart]),I).}
\]

\[
\text{occurs\_process(instantStart,I):-}
\text{happens\_over(par(releaseDisableButton,begin(burning)),I).}
\]

\[
\text{occurs\_process(delayedStart,I):-}
\text{happens\_over(seq(releaseDisableButton,begin(burning)),I),}
\text{iholds\_for(thermostaticValveClosed,I), iholds\_for(pilotLightOn,I),}
\text{iholds\_for(powerSupplied,I), iholds\_for(releasedDisableButton,I).}
\]

\[
\text{occurs\_process(heatingSequence,I):-}
\text{happens\_over(pseq(heating,N),I), iholds\_for(pilotLightOn,I),}
\text{iholds\_for(powerSupplied,I), iholds\_for(releasedDisableButton,I).}
\]

\[
\text{occurs\_process(heating,I):-}
\text{happens\_over(seq(begin(burning),end(burning)),I),}
\text{iholds\_for(burning,I).}
\]

\[
\text{occurs\_process(endHeatingCycle,I):-}
\text{happens\_over(salt([instantEnd,delayedEnd]),I).}
\]
occurs_process(instantEnd,I):-
  happens_over(par(end(burning),
                walt([pilotLightExtinction, pressDisableButton,
                      closeGasMain, switchOffPower])),I).

occurs_process(delayedEnd,I):-
  happens_over(seq(end(burning),
                 walt([pilotLightExtinction, pressDisableButton,
                      closeGasMain, switchOffPower])),I),
  iholds_for(thermostaticValveClosed,I), iholds_for(pilotLightOn,I),
  iholds_for(powerSupplied,I), iholds_for(releasedDisableButton,I).

With respect to the previously illustrated database, a query consisting of a
totally unbound occurs_process gives the following solutions:

?- occurs_process(X,C).

X = heatingCycle,  X = startHeatingCycle,  
  C = [6,19]         C = [6,8]              

X = delayedStart,  X = heatingSequence,  
  C = [6,8]         C = [8,10]             

X = heatingSequence,  X = heatingSequence,  
  C = [11,19]         C = [8,19]           

X = heating,  X = heating,  
  C = [8,10]         C = [11,19]           

X = endHeatingCycle,  X = instantEnd,  
  C = [19,19]         C = [19,19]           

EVALUATION OF EXPERIMENTAL RESULTS

In the following we highlight some of the computational problems we encoun-
tered in experimenting the logical calculus with an ordinary Prolog inter-
preter to execute models. The problems fall in two categories: (i) queries
plagued by looping problems; (ii) inefficiency due to redundant computa-
tion.

'Naive' domain or system representations can't be directly run in Prolog
because they do not take into account problems such as looping problems.
Even if they are executable, their computational properties are almost al-
ways unsatisfactory, because Prolog performs a great deal of redundant
computation, and the execution results unnecessarily slow. To overcome
these problems, different strategies have been identified [17]. They basically consist of more sophisticated theorem provers involving lemma storage, delayed evaluation, etc, transformations of the language that give it better computational properties with respect to Prolog, and compilations of models into executable representations such as timed Petri Nets, extended finite state automata, etc. A first solution consists of leaving the axioms as they are and using a different theorem prover to Prolog, one which incorporates loop-checking, constraint processing, delayed evaluation, and so on. A second solution consists of sticking with Prolog as the theorem prover and transforming the axioms so that they can be executed with ordinary Prolog. A third solution consists of providing some hybrid of the previous two strategies, that is, a combination of sophisticated theorem proving and transformation technologies. Finally, one can abandon the theorem proving approach altogether, and construct a suite of tailor-made algorithms which perform the same operation or compile the logical model into a number of operational models, one for each of the relevant modes according to which goals can be entered, that can be directly executed. Elsewhere, we tried to face these problems by building different theorem provers to Prolog [8]; in this paper, we experimented the alternative approach of keeping Prolog as the theorem prover and changing the axioms.

**The problem of feedback loops**

During modeling of the gas heater, it became apparent that the calculus (as it is) is unable to manage loops in the model, as in the case of feedback loops in physical systems: “Feedback occurs when a sequence of cause-effect interactions produces an effect on antecedents in the sequence. There is no new information to be gained by propagating this feedback signal around the loop again (ad infinitum)”[6].

For example, in the gas heater a feedback loop exists between the pilot light, the Safety System and the Safety Valve: the pilot light can be on if the Safety Valve is open; the safety Valve is open only if the Safety System senses that the pilot light is on. By the way, this is the reason why the designer of the system introduced the Safety Disable Button in order to start up the pilot light when it is off.

If the model of the feedback loop involves context-dependent initiates or terminates, a loop occurs when these clauses come into play during the computation.

In order to overcome this problem, we have adopted as a first resort a principled approach to model building in order to break the loops since we want to keep the Prolog theorem prover unchanged. We adopted the principle that feedback loops must be broken by substituting one of the context dependent initiates.at or autoinitiates.at (with its corresponding terminates.at or autoterminates.at) clauses which is part of the loop with a suitable pair of events which respectively initiates and terminates the
corresponding property. Choice of this pair can not be arbitrary, but it must refer to events which are easily observable/measurable so that the possibility of monitoring them in a real-world setting is plausible. However, it should be noted that this approach, while on one hand increases efficiency of execution by avoiding the introduction of additional machinery in the interpreter to solve the looping problem, on the other hand has the drawback of imposing an heavy burden on the modeler. We are thus investigating an alternative approach consisting in the addition of a loop checker to the reasoning mechanisms in order to test whether a query we are trying to prove reappears in the resolvent.

Efficiency issues
Even considering the calculus of events only, computational inefficiency soon arises if we consider a realistically large database to which the same queries are frequently posed. Considering the case of a \text{mholds.\_for} query with only the property argument instantiated and assuming to reason in a model where all \text{initiates.\_at} and \text{terminates.\_at} are context-independent, the complexity of the query (measured using access to \text{happens.\_at} facts as a reference) is cubic in the number of initiating and terminating events for that property. If indeed we have \( n \) initiating events with their corresponding \( n \) terminating events in the database, during the exploration of the search tree for the partially bound \text{mholds.\_for} query, the two \text{happens.\_at} predicates in the \text{mholds.\_for} definition will have the temporal argument unbound; as a result, every possible pair of initiating and terminating event (\( n^2 \) accesses to \text{happens.\_at} facts) will be picked up; the number of these pairs that will pass the End > Start test stays quadratic since it will be \( n + (n - 1) + \ldots + 1, \text{i.e.} (n(n + 1)/2) \). A \text{not broken.\_during} query will be invoked for each of the survived pairs; complexity of this query is linear, the worst case given by pairs of a initiating event immediately followed by its corresponding terminating event (all \( n \) instances of terminating events will be checked against the pair). As a consequence, the total number of accesses to \text{happens.\_at} facts will be cubic.

Addition of context independency to the model heavily deteriorates computational performance. Let \( P(> 0) \) be the maximum number of preconditions in a clause, \( n \) the maximum number of initiating-terminating pair of events for a single property and \( B \) the maximum number of backward directions from a single precondition. The order of complexity of the query (measured using access to \text{happens.\_at} facts as a reference) is now \( O(n^{P+1}) \). If we indeed consider \( B = 0 \), we fall in the already discussed case of the event calculus without preconditions, where the complexity is \( n^3 \). What additionally happens when \( B \) is 1 or more is that after each of the \( n^3 \) accesses to an \text{happens.\_at}, during the exploration of the search tree for the partially bound \text{mholds.\_for} query, at worst \( P \) preconditions are checked: each \text{holds.\_at} query will result in a call to a \text{mholds.\_for} predicate with the temporal argument
Artificial Intelligence in Engineering

unbound and a maximum of $B - 1$ backward indirections. So, the complexity will become $COM(P, B) = P \cdot COM(P, B - 1) \cdot n^3$. Since $COM(P, 0) = n^3$, it is easily provable by induction that $COM(P, B) = P^B \cdot n^{(B+1)3}$.

However, a careful examination of the search tree reveals that this undesired increase in complexity is largely due to redundant computation of the mholds_for queries called during precondition verification. This suggests that a caching mechanism should benefit the event calculus and increase its applicability. Caching would allow to store results of mholds_for queries for use both in answering later queries and in avoiding computation of multiple identical queries in the current proof. However, creating a proper caching mechanism is not a trivial task. One main issue to confront will be the management of updates to the database: the easy solution of rebuilding the cache from scratch at every update is unacceptable even for small databases. The caching system should be able to incrementally add new results to the old ones and to update or delete only the old results which need it.

CONCLUSIONS

The paper presented a logical calculus that extends the Event Calculus with high-level primitives for context dependency and discrete process modeling. This is the first building block of a general and efficient calculus dealing with discrete, continuous and abnormal processes in a uniform logical framework. It has been implemented on a Sun SPARC2 with Quintus Prolog and its effectiveness has been tested by producing an executable specification of a gas heater system.

REFERENCES