A Realistic Experiment in Knowledge Representation in Open Event Calculus: Protocol Specification

Marc Denecker¹ Kristof Van Belleghem² Guy Duchatelet³ Frank Piessens⁴ Danny De Schreye⁵
K.U.Leuven, Department of Computing Science
Celestijnenlaan 200A, 3001 Heverlee, Belgium
e-mail: {marcd, kristof, frank, dannyd}@cs.kuleuven.ac.be, gudu@telinfo.be

Phone: ++32 16 327544
Fax: ++32 16 327996

Keywords: Knowledge Representation, Temporal Reasoning, Extensions of Logic Programming

Abstract

This paper presents one of the first realistic experiments in the use of Event Calculus in Open Logic Programming: the specification of a process protocol. The specification task involves most of the common complications of temporal reasoning: the representation of context dependent actions, of preconditions and ramifications of actions, the modelling of system faults, and most of all, the representation of uncertainty of actions. As the underlying language, the Open Logic Programming formalism, an extension of Logic Programming, is used. The experiment shows that Event Calculus is a promising candidate for the specification of dynamic systems. A comparison between specification of process protocols in Event Calculus and in the more commonly used process algebras shows fundamental differences between the two approaches.

¹ supported by the K.U.Leuven research coordination
² supported by the Belgian IWT
³ Telindus Distribution
⁴ supported by the Belgian National Fund for Scientific Research
⁵ senior research associate of the Belgian National Fund for Scientific Research
1 Introduction

In the past a number of experiments have been carried out to study the usefulness of extensions of logic programming for specification and reasoning on temporal domains ([11], [14], [21], [9], [10], [7], [20],[23],[24]). Although successful, many of these experiments have a fairly academic flavour. In this paper, we conduct a more realistic experiment, using open logic programming and Event Calculus for the specification of a non-trivial temporal domain. The sliding window protocol with go-back-n [22] is a well-known process protocol in the area of communication protocols. Currently, the most important formal languages for the specification of process protocols are the process algebras: CSP [12], CCS [18] and their descendants such as LOTOS [1], etc. These languages are based on a mathematical abstraction of a process as an algebraic entity which can be constructed by combining more basic processes using a class of pre-defined operators.

On the other hand, a distributed system of processes is just one example of a dynamic system of entities. Specification of and reasoning on dynamic systems is the aim of the temporal reasoning area in Artificial Intelligence. Until fairly recently, the major obstacle in specifying dynamic systems and temporal domains in logic was the frame problem: the problem of giving a precise description of the effects of an action on the state of the world [17]. Since the beginning of the decade, this problem is gradually being solved in Situation and Event calculus [14]. In [5], we presented a simple but expressive form of event calculus and showed its use for representing various forms of uncertainty, including uncertainty on the events, indeterminate actions, unknown initial state, etc. Event Calculus has been developed as a universal theory to describe actions and change. In contrast to process algebras, where the concept of a process is hardcoded in the semantics, Event Calculus does not make any ontological assumption about the nature of a process. In our specification, a process is modeled as a dynamic entity with an identity (a name) and an internal state which is represented by a set of attributes or relations. A process has a restricted ability to sense actions executed by other entities (e.g. when receiving or synchronising on messages) and has the ability to execute actions (e.g. by sending messages). Both sensing and executing actions modify its internal state and a restricted part of the outer world (e.g. receiving a message deletes a message from a channel). In section 3, we adopt this dynamic view on processes in a specification of the sliding window protocol.

We believe that our study is interesting from different points of view. First, it gives a non-trivial application of open logic programming for specification, and clarifies our view on this formalism. Second, it shows that the temporal reasoning theories developed in artificial intelligence are growing out of their infancy. Third, the representation style of processes in the experiment is strikingly different from that in a process algebra specification. As argued in section 5, this is due to the differences between the algebraic view on processes and the view of a process as a dynamic entity with state and identity. The representation style we adopt will be argued to be more generally applicable, and therefore more suited for integration with other applications.

Though the view of a process as a dynamic entity fits in more naturally with the ontological primitives of Event Calculus, the Event Calculus is sufficiently general to also represent the algebraic view on processes. This makes an integration of process algebra specifications in Event Calculus feasible. Section 4 presents a general technique to incorporate protocol specifications using process algebras in Event Calculus.
2 Open Event Calculus

As the underlying logic for the specification, we use the logic of open logic programs (\cite{5}), an integration of logic programming (\cite{16}) and classical logic. An open logic program is a pair \((P^D, T)\) such that \(P\) is set of program clauses: implications of the form \(A \leftarrow B\) where \(A\) is an atom of the form \(p(t_1, \ldots, t_n)\) (with \(t_1, \ldots, t_n\) terms) and \(B\) is a conjunction of literals of the form \(p(t_1, \ldots, t_n)\) or \(\neg p(t_1, \ldots, t_n)\). All predicates which occur in the head of a rule of a program clause belong to the set \(D\) of defined predicates; \(D\) may contain other predicates with an empty definition. These predicates are always false. The other predicates are called open predicates. \(T\) is a set of classical first order logic axioms.

Our logic is a terminological interpretation of the formalism of abductive logic programs. \(P^D\) is to be considered as a set of definitions for the predicates in \(D\). The clauses of a logic program define the predicates of \(D\) by exhaustively enumerating the cases in which the predicates are true. This is reflected in the formal semantics of this formalism. Under the Console completion semantics (\cite{4}), the meaning of an open logic program \((P^D, T)\) is given by the first order logic theory consisting of \(T\) augmented with Clark’s Equality Theory and the completed definitions of all defined predicates in \(P^D\) (\cite{3}). Clark’s Equality Theory expresses that different terms represent different objects. The completed definition of a defined predicate \(p \in D\) is obtained as follows. Assume that \(\{p(t_1) \leftarrow B_1, \ldots, p(t_m) \leftarrow B_m\}\) is the set of all program clauses of \(P\) with \(p\) in the head. Here \(t_i\) represents a tuple of terms \((t_1^i, \ldots, t_n^i)\). Assume that \(\bar{X}_i\) is the tuple consisting of all variables occurring in \(p(t_i) \leftarrow B_i\). Then the completed definition of \(p\) with respect to \(P^D\) is the equivalence:

\[
\forall \bar{X} : p(\bar{X}) \leftrightarrow (\exists \bar{X}_1. \bar{X} = \bar{t}_1 \land B_1) \lor \ldots \lor (\exists \bar{X}_m. \bar{X} = \bar{t}_m \land B_m)
\]

In logic programming, normally all predicates are completed. In an open logic program, the open predicates are not completed. Their truth value is not determined by the program, thus making it possible to represent incompletely known domains. In many cases, the expert will have partial domain knowledge which does not define any predicate. This assertional knowledge can be represented by FOL axioms in \(T\).

The use of the open logic program formalism (OLP) for representing definitional knowledge and assertional knowledge is well illustrated in the Event Calculus. An Event Calculus is an open logic program which describes a world evolving in time due to events which affect the state of the world. A fundamental assumption in event calculus is that the state of the world at any point in time is determined completely by the initial state and the effects of actions in the past. Informally, a property or fluent \(P\) holds at a certain moment \(T\) if it is initiated by some earlier event \(E\) and is not terminated between \(E\) and \(T\), or if \(P\) holds initially and is not terminated before \(T\). This knowledge is formulated as the following definition for the predicate \(\text{holds\_at}(P, T)\):

\[
\begin{align*}
\text{holds\_at}(P, T) & \leftarrow \text{happens}(E), E < T, \text{initiates}(E, P), \neg \text{clipped}(E, P, T) \\
\text{holds\_at}(P, T) & \leftarrow \text{initially}(P), \neg \text{clipped}(P, T) \\
\text{clipped}(E, P, T) & \leftarrow \text{happens}(C), E < C < T, \text{terminates}(C, P) \\
\text{clipped}(P, T) & \leftarrow \text{happens}(C), C < T, \text{terminates}(C, P)
\end{align*}
\]

\(\text{clipped}(E, P, T)\) is an auxiliary predicate which intuitively means that the property \(P\) is terminated during the interval \([E, T[\). \(\text{clipped}(P, T)\) means that the property \(P\) is terminated before \(T\). In Event Calculus, \(<\) represents a linear order on the events. In addition, for the
purposes of this paper, we can assume that an event is associated with at most one action. This knowledge is formulated in the following FOL axioms:

\[
\begin{align*}
X < Y \land Y < Z & \rightarrow X < Z & \text{transitivity} \\
\rightarrow X < Y, Y < X & \rightarrow X < Z & \text{asymmetry} \\
happens(X) \land happens(Y) & \rightarrow X < Y \lor X = Y \lor Y > X & \text{linearity} \\
act(E, A1) \land act(E, A2) & \rightarrow A1 = A2 & \text{act}
\end{align*}
\]

The definitions of holds_at and clipped rely on the predicates *initially*, *happens*, <, *initiates* and *terminates*, which describe the initial situation, the events, their order and their effects. Depending on the given information about the domain, different subsets of these predicates will be defined. Typically, both *initiates* and *terminates* are defined predicates, defined by rules of the form:

\[
\text{initiates}(E, < \text{fluent} >) \leftarrow \text{act}(E, < \text{action} >), < \text{Conditions} >
\]

which represents that an event \( E \) on which an action happens, initiates (or terminates) a fluent if some conditions hold on the event \( E \). Another important form of knowledge in the context of temporal domains are *action preconditions*: conditions which have to be satisfied in order to let some action take place. For example, a robot can put a block down on the table only if it is holding the block. We generalize and formalize this type of constraint in the following FOL axiom:

\[
happens(E) \land \text{act}(E, A) \rightarrow \text{precondition}(A, E)
\]

*precondition*\( (A, E) \) means that a precondition for the action \( A \) to happen on event \( E \) is satisfied. The predicate *precondition* will be defined by problem specific rules.

A last comment is on the representation of complex objects. Modeling a complex object in logic can be done by defining a number of predicates representing the relations in which the object may occur. This can be opposed to the object-oriented style which uses *attributes* to represent the knowledge on an object. Attributes are suitable to represent partial functions of complex objects and time. We have often found it useful and elegant to use a mixed representation for complex objects, using attributes for the partial functions, and using predicates for other types of relations. Attributes are described by the fluent predicate *attribute*/3. The atom

\[
\text{holds}\_at(\text{attribute}(P, PROPERT\text{Y}, VAL), T)
\]

means that the entity \( P \) has a value \( VAL \) for attribute \( PROPR \), at time \( T \). This representation allows us to formulate the law of destructive assignment for all attributes: any initiation of a value for an attribute deletes the old value:

\[
\begin{align*}
\text{terminates } & (E, \text{attribute}(P, PROPERT\text{Y}, OLD\_VALUE)) \\
\rightarrow & \text{initiates } (E, \text{attribute}(P, PROPERT\text{Y}, VAL)), \\
\text{holds}\_at & (\text{attribute}(P, PROPERT\text{Y}, VAL), E), \\
\neg & \text{OLD\_VALUE} = VAL
\end{align*}
\]

This axiom is assumed in the specification below.
3 Specification of a sliding-window protocol

In this section, we give an example of a specification of a non-trivial protocol in Event Calculus: the sliding-window protocol with go-back-n. This is a well-known communication protocol, described in [22], which is situated in the OSI datalink-layer. The protocol provides a reliable and quite efficient connection to be used by the higher OSI layer, the network layer. The network layer passes frames down, where the sliding-window protocol makes use of the services of the physical layer to pass them to the peer on the other side. It is a symmetrical protocol which uses pipelining and piggybacked acknowledges, combined with flow control based on a sending window of sent unacknowledged frames and a receiving window (of length 1) of acceptable frames.

We give an informal description of the protocol. Essentially, a process in the datalink layer iterates the following actions. Initially the process is waiting for an input event. Three types of events are possible:

- a send event: the network-layer transmits a frame to the datalink process. The latter process puts the frame in a slot in its sending window, initiates the timer associated with that slot, and sends the frame, with some additional information included, out on the physical channel. This additional information consists of the number of the sent frame's slot and an acknowledgement indicating the number of the last received frame.

- a receive event: a packet is received from the physical channel. If it does not contain an expected frame (i.e. its slot number is not the successor of the last received packet's) or if the packet is corrupted, the packet is dropped. Otherwise, the packet is accepted and the frame it carries is transmitted to the network layer. Also, the acknowledgement number carried by the packet indicates that all frames up to that number are received by the peer process, thus all slots in the sending window up to the acknowledge number can be cleared and all associated timers turned off.

- a timer event: the timer of (the oldest) frame in the window is ringing. The process goes into a mode in which it retransmits its buffer over the channel. After this is done, the process waits again for an input event.

In the following sections, this informal specification will be used as a guideline for a formal specification in event calculus. In this specification the predicates \textit{happens}/1, \textit{act}/2 and \textit{<}/2 will be open. This is because a protocol specifies only \textit{potential} behaviour: the protocol only specifies which events \textit{can} occur. This implies that we cannot give a definition for these predicates.

3.1 World description

We now describe and specify the environment in which the protocol "operates": the main types of objects in the domain, their attributes and relations. The main type of objects are the datalink processes, the channel, the frames and of course the events which take place. There are two peer-processes (both are sender/receiver), which are connected by a channel. The two processes are denoted \( p_1 \) and \( p_2 \). \( p_1, p_2 \) are each others receiver processes. The following clauses represent this:

\[
\text{process}(p1)
\]
process(p2)  
receiver(p1, p2)  
receiver(p2, p1)

The attributes of a process are listed below:

- **mode**: this attribute can take four different values:
  - *input*: the process is ready and waiting for input
  - *sending*(F, NR): the process is sending a frame F stored in the slot NR
  - *receiving*(F): the process is transmitting a just received frame F to its network layer
  - *retransmitting*(NR): after a time-out, the process is retransmitting its buffer starting from slot number NR.

- **xpf**: the *expected frame* attribute: this attribute points to the number of the next frame that is expected.

- **i** (for any \(0 \leq i < n\) with \(n\) the size of the window): the value of the attribute \(i\) of process \(P\) is the frame which is stored at slot \(i\) of the window. The window is a circular buffer of length \(n\). This is represented by the predicate \(cnext\) which is defined as follows:
  
  \[
  cnext(0, 1) \\
  \ldots \\
  cnext(n - 1, 0)
  \]

- **xpa**: the *expected acknowledge* attribute: this attribute takes values \(0 \leq i < n\) and points to the oldest sent but unacknowledged frame in the window.

- **fts**: the *frame to send* attribute: this attribute takes values \(0 \leq i < n\) and points to the first free slot in the window.

A time-dependent relation on processes is \(networklayer\_enabled(P)\): It means that the network-layer of process \(P\) is enabled to give new frames to \(P\).

We distinguish between frames and packets. A frame is an *occurrence* of a data unit given by network-layer to the data-link process. A packet is an *occurrence* of data which is physically sent over the channel. Two different frames or two different packets may carry precisely the same information. Yet we need to distinguish between packets and frames carrying the same information but created at different occasions in order to express elegantly some of the physical properties of the system. Take, for example, the property that if one packet is received earlier than another packet, then the first packet was sent earlier than the second. If in this sentence, a packet would be interpreted as the piece of data rather than the occurrence, then one packet might appear many times on the channel. The above statement would then state that if some piece of data A is received earlier than another piece of data B, then there is at least one event of sending A which occurs earlier than at least one event of sending B. This is a far weaker statement than the intended one.

How do we represent frames and packets? A frame is introduced in the system at the event \(E\) when the network layer of a process \(P\) gives the frame to \(P\). We denote the frame by the term \(frame(E, P)\). We denote a packet by \(packet(E, FRAME, NR, ACK)\). \(E\) is the
send event at which the packet is created and sent over the channel. FRAME, NR, ACK are the data carried in the packet. FRAME is the carried frame, NR the slot in which the frame is stored and ACK the slot of the last successfully received frame. The slot NR of the frame is used by the receiver to identify the frame.

Packets appear in two different relations:

- *corrupt(PACKET)*: the packet PACKET is corrupt (due to errors of the channel);
- *on_channel(PACKET, P)*: the packet PACKET is being sent to the receiver P.

There are 7 types of events. The first four model the sending and receiving actions by peer-processes and their network layers. One action models a timer run-out. The latter two model two different errors of the channel:

- *net_send(F, P)*: the network layer gives the frame F to its peer process P.
- *net_receive(P, F)*: the peer process P passes a received frame F to its network layer;
- *send(P, PACKET)*: P sends packet PACKET out on the channel.
- *receive(P, PACKET)*: P receives packet PACKET from the channel.
- *timer_rings(P, NR)*: the timer of slot NR of process P rings;
- *disturbance_channel(PACKET)*: a disturbance of the channel corrupts packet PACKET;
- *failure_channel(PACKET)*: a failure of the channel causes the loss of the packet PACKET.

### 3.1.1 Initial state

The following clauses define the initial state, in which both processes are in input mode, have an empty window, and have enabled network layers:

\[
\begin{align*}
\text{initially(attribute(p1, mode, input))} \\
\text{initially(attribute(p2, mode, input))} \\
\text{initially(attribute(p1, fts, 0))} \\
\text{initially(attribute(p1, xpa, 0))} \\
\text{initially(attribute(p1, xpf, 0))} \\
\text{initially(attribute(p2, fts, 0))} \\
\text{initially(attribute(p2, xpa, 0))} \\
\text{initially(attribute(p2, xpf, 0))} \\
\text{initially(networklayer_enabled(p1))} \\
\text{initially(networklayer_enabled(p2))}
\end{align*}
\]
3.1.2 Channel

We start by describing the channel. A fundamental assumption about the physical channel is that it preserves the order of packets. The following FOL axiom formulates this:

\[
\begin{align*}
false \leftarrow & \ receiver(PROCESS1, PROCESS2), \\
& \ act(E3, receive(PROCESS2, PACKET1)), \\
& \ act(E4, receive(PROCESS2, PACKET2)), \\
& \ E4 < E3 \\
\end{align*}
\]

Two types of events are local to the channel. These are the events which simulate errors of the channel. Their preconditions and effects are described below:

- A disturbance on the channel corrupts a packet. A precondition for this to happen is that the frame is on the channel:

  \[
  \begin{align*}
  & \ precondition \left( disturbance \_channel \ (PACKET), E \right) \\
  \leftarrow & \ holds\_at \ (on\_channel \ (PACKET, PROCESS), E) \\
  \end{align*}
  \]

  Its effect is that it initiates `corrupt`:

  \[
  \begin{align*}
  & \ initiates(E, corrupt(PACKET)) \\
  \leftarrow & \ act(E, disturbance\_channel \ (PACKET)) \\
  \end{align*}
  \]

- A failure of the channel which causes the loss of a packet. Precondition is that the packet is on the channel:

  \[
  \begin{align*}
  & \ precondition \left( failure \_channel \ (PACKET), E \right) \\
  \leftarrow & \ holds\_at \ (on\_channel \ (PACKET, PROCESS), E) \\
  \end{align*}
  \]

  Its effect is that it terminates the packet’s being on the channel:

  \[
  \begin{align*}
  & \ terminates(E, on\_channel \ (PACKET, PROCESS)) \\
  \leftarrow & \ act(E, failure\_channel \ (PACKET)) \\
  \end{align*}
  \]

3.1.3 The sending behavior of a peer process

For the description of a peer process, we follow the structure of the informal specification of the protocol in section 3. Assume that the process receives a frame from the network layer. Preconditions for this event to happen are that the peer process is in input mode, that the network layer is enabled and that the frame sent, is the frame uniquely corresponding to the `net\_send` event and the process.

\[
\begin{align*}
& \ precondition \left( net\_send \ (frame(E, PROCESS), PROCESS), E \right) \\
\leftarrow & \ process(Process), holds\_at \ (attribute \ (PROCESS, mode, input), E), \\
& \ holds\_at \ (networklayer\_enabled \ (PROCESS), E) \\
\end{align*}
\]
The effect of a `net_send(FRAME, PROCESS)` event is that the process enters sending mode, that the frame is stored in the first free slot of the window, as indicated by the `fts` (frame to send) attribute, that `fts` is incremented, and that if the buffer is full, the network-layer is disabled.

\[
\text{initiates } (E, \text{ attribute } (\text{PROCESS, mode, sending (FRAME), FTS})), \\
\text{initiates } (E, \text{ attribute } (\text{PROCESS, FTS, FRAME})), \\
\text{initiates } (E, \text{ attribute } (\text{PROCESS, fts, NEXTFTS})), \\
\text{act } (E, \text{ net_send (FRAME, PROCESS)}), \\
\text{← holds_at (attribute ) (PROCESS, fts, FTS), E}, \\
\text{cnext (FTS, NEXTFTS)}
\]

As mentioned, the network layer is disabled when the peer is out of buffer space. In fact, it must be disabled even sooner, when there is only one free buffer slot left (see [22] for more details). There is only one buffer slot left when `fts` points to the predecessor of `xpa`. This is expressed by:

\[
\text{terminates } (E, \text{ networklayer enabled (PROCESS)}), \\
\text{initiates } (E, \text{ attribute } (\text{PROCESS, fts, FTS})), \\
\text{← holds_at (attribute ) (PROCESS, xpa, XPA), E}, \\
\text{cnext (FTS, XPA)}
\]

An implicit effect of the `net_send` event is that the timer of the slot of the frame is set. Here we simply assume that the timer of a slot is set as long as the slot is in the window; or, from the moment a frame is stored in the slot until an acknowledge for the slot is received.

The only event that a process can execute when it is in sending mode is to send the frame, together with its slot number and an acknowledge. The precondition of the send event is that the process should be in sending mode and that the packet carries the correct data (frame, slot number and acknowledgement), and that the packet is the one uniquely corresponding to this event:

\[
\text{precondition } (\text{send (PROCESS, packet(E, FRAME, NR, ACK), E)}, \\
\text{process(PROCESS)}, \\
\text{← holds_at (attribute ) (PROCESS, mode, sending(FRAME, NR)), E}, \\
\text{holds_at (attribute ) (PROCESS, xpf, XPF), E}, \\
\text{cnext (ACK, XPF)}
\]

An obvious effect of sending is that the packet is on the channel.

\[
\text{initiates } (E, \text{ on_channel (PACKET, RECEIVER))}, \\
\text{act } (E, \text{ send (PROCESS, PACKET)}), \\
\text{receiver(PROCESS, RECEIVER)}
\]

Another effect of sending in this mode is that the process returns to input mode.

\[
\text{initiates } (E, \text{ attribute (PROCESS, mode, input))}, \\
\text{act } (E, \text{ send (PROCESS, PACKET)}), \\
\text{holds_at (attribute ) (PROCESS, mode, sending(FRAME, NR)), E},
\]

8
### 3.1.4 The receiving behaviour of peer-process

When the peer process is in input mode, it may receive packets from the channel. Precondition for this event to happen is that the peer process is in input mode and that the packet is on the channel:

```plaintext
precondition (receive (PROCESS, PACKET), E) 
holds_at (attribute (PROCESS, mode, input), E),
holds_at (on_channel (PACKET, PROCESS), E),
```

The effect of this event depends on several factors and is split up in different rules. One effect of receiving is always that the frame is removed from the channel:

```plaintext
terminates (E, on_channel (PACKET, PROCESS)) 
act (E, receive (PROCESS, PACKET)),
```

To be accepted, a received packet should not be corrupted and should carry the expected frame number given by attribute \( xpf \). If these conditions are satisfied, the \( xpf \) attribute is circularly increased and the process enters the receiving mode during which the frame is passed to the network layer.

```plaintext
initiates (E, attribute (PROCESS, xpf, NEXT XPF))
initiates (E, attribute (PROCESS, mode, receiving(FRAME)))
act (E, receive (PROCESS, PACKET)),
¬ holds_at (corrupt (PACKET), E)
PACKET = packet(E', FRAME, NR, ACK),
holds_at (attribute (PROCESS, xpf, XPF), E), NR = XPF, cnext (XPF, NEXT XPF)
```

A second part of the receiving behaviour is the treatment of the acknowledgement. When an uncorrupted packet is received and it carries an acknowledgement \( ACK \) which corresponds to some currently used slot of the sending window, then all slots from the expected acknowledgement \( xpa \) up to the slot \( ACK \) are released by setting the \( xpa \) attribute to the successor of \( ACK \). Due to the nature of the protocol, this is safe even when the received acknowledgement \( ACK \) is not the expected acknowledgement.

In addition, if the network layer was disabled, it will now be enabled again. This is formalized in the following clauses.

```plaintext
initiates (E, attribute (PROCESS, xpa, NEXT XPA))
  | initiates (E, network_layer_enabled (PROCESS))
  | act (E, receive (PROCESS, PACKET)),
  | holds_at (on_channel (PACKET, PROCESS), E),
  | ¬ holds_at (corrupt (PACKET), E),
  | PACKET = packet(E', FRAME, NR, ACK),
  | holds_at (attribute (PROCESS, xpa, XPA), E),
  | holds_at (attribute (PROCESS, fts, FTS), E),
  | c_in_between (XPA, ACK, FTS),
  | cnext (ACK, NEXT XPA)
```
The predicate $c \in \text{between}(X, I, Y)$ denotes that $I$ is circularly in between $X$ and $Y$. It can be easily defined as follows:

$$
c \in \text{between}(X, I, Y) \leftarrow X \neq Y
$$

$$
c \in \text{between}(X, I, Y) \leftarrow X \neq Y, c\text{next}(X, X1), c \in \text{between}(X1, I, Y)
$$

When a process is in receiving mode, it gives the received frame to the network layer. The precondition of the action $\text{net\_receive}$ is that the process is in receiving mode.

$$
\text{precondition (net\_receive (PROCESS, FRAME), E)}
\leftarrow \text{holds\_at (attribute (PROCESS, mode, receiving(FRAME)), E)}
$$

The effect of $\text{net\_receive}$ is that the process returns to input mode:

$$
\text{initiates (E, attribute (PROCESS, mode, input))}
\leftarrow \text{act (E, net\_receive (PROCESS, FRAME))}
$$

### 3.1.5 Handling of a ringing timer

Next we show how ringing timers are handled. We know that whenever a timer rings, the buffer is non-empty and, since all timers have an equal timing interval, the timer corresponds to the oldest frame in the sending window. This yields the following precondition for $\text{timer\_rings}$:

$$
\text{precondition (timer\_rings (PROCESS, XPA), E)}
\leftarrow \text{process(PROCESS),}
\text{holds\_at (attribute (PROCESS, mode, input), E)},
\text{holds\_at (attribute (PROCESS, xpa, XPA), E)},
\text{holds\_at (attribute (PROCESS, fts, FTS), E)},
\neg XPA = FTS
$$

When this timer rings, all unacknowledged frames have to be retransmitted. The process enters the retransmitting mode. The attribute $\text{retransmit}$ is initialised to the expected acknowledge.

$$
\text{initiates (E, attribute (PROCESS, mode, retransmitting(XPA)))}
\leftarrow \text{act (E, timer\_rings (PROCESS, XPA))}
$$

While a process is in retransmitting mode, it sends out its buffer. This situation yields another precondition for a send event: the process is in retransmitting mode and the sent frame is the one pointed to by the $\text{retransmit}$ attribute:

$$
\text{precondition (send (PROCESS, packet(E, FRAME, NR, ACK)), E)}
\leftarrow \text{holds\_at (attribute (PROCESS, mode, retransmitting(NR)), E)},
\text{holds\_at (attribute (PROCESS, NR, FRAME), E)}
\text{c\text{next} (ACK, XPF)}
$$

The effects of sending in retransmitting mode are slightly different than in sending mode. The additional effect is that either the slot number to be retransmitted is incremented or,
when the whole buffer is retransmitted, that the process returns to input mode:

\[
\text{initiates} \ (E, \text{attribute} \ (\text{PROCESS}, \text{mode}, \text{retransmitting}(\text{NEXTNR})))
\]

\[
\text{act} \ (E, \text{send} \ (\text{PROCESS}, \text{PACKET}), \text{hold at} \ (\text{attribute} \ (\text{PROCESS}, \text{mode}, \text{retransmitting}(\text{NR})), E),
\]

\[
\text{cnext} \ (\text{NR}, \text{NEXTNR}), \text{hold at} \ (\text{attribute} \ (\text{PROCESS}, \text{fts}, \text{FTS}), E),
\]

\[
\neg \text{FTS} = \text{NEXTNR}
\]

\[
\text{initiates} \ (E, \text{attribute} \ (\text{PROCESS}, \text{mode}, \text{input}))
\]

\[
\text{act} \ (E, \text{send} \ (\text{PROCESS}, \text{PACKET}), \text{hold at} \ (\text{attribute} \ (\text{PROCESS}, \text{mode}, \text{retransmitting}(\text{NR})), E),
\]

\[
\text{cnext} \ (\text{NR}, \text{FTS})
\]

This concludes the specification of the sliding window protocol with go-back-n.

4 Translating other specification formalisms to Event Calculus

In this section, we show how any specification formalism whose semantics can be defined in terms of a labeled transition system by means of a logic program, can easily be translated to the Event Calculus. In particular, we show that process algebras like LOTOS, CCS and CSP can be translated to the Event Calculus.

The purpose of such a translation is twofold. First, it shows that the Event Calculus has at least the expressive power of process algebras. Secondly, the existence of such a straightforward translation makes the process of combining specifications in different languages easier: if one is building an Event Calculus specification, and wants to reuse existing specifications written in a process algebra, these existing specifications can be incorporated in the Event Calculus specification by translating them as explained in this section. Of course, the translated specifications will not be very well-structured from an Event Calculus point of view, but all existing tools for the Event Calculus can be used on them.

A labeled transition system consists of two sets (a set \(S\) of states and a set \(L\) of transition labels), and a relation on \(S \times L \times S\), called the transition relation.

Labelled transition systems are widely used to give a semantics to specification formalisms for concurrent systems. Usually, \(L\) is interpreted as a set of actions, and \((s_1, l, s_2)\) is in the transition relation if the specified system can go from state \(s_1\) to state \(s_2\) by doing action \(l\). For example, the semantics of LOTOS ([1]) is defined by means of a labeled transition system, where \(S\) is the set of possible behaviour expressions and \(L\) is the set of possible actions.

A labeled transition system can be defined in a logic program by defining a predicate \textit{transition/3}. For example, we can define the semantics of Basic LOTOS behaviour expressions, built on the constant \texttt{stop}, the prefix operator \texttt{';}, the choice operator \texttt{''} and the parallel composition operator \texttt{''|} (where \(S\) is a list of gates) by the following logic program:

\[
\text{transition}(A; B, A, B).
\]

\[
\text{transition}(B_1[[B_2], A, B_1') \leftarrow \text{transition}(B_1, A, B_1').
\]

\[
\text{transition}(B_1[[B_2, A, B_2') \leftarrow \text{transition}(B_2, A, B_2').
\]
It is straightforward to define the transition predicate for other process algebra connectives.

Once we have defined the transition predicate, translation to Event Calculus is easy. Suppose we have an arbitrary labeled transition system defined by a predicate \( \text{transition}(\text{STATE}_1, \text{ACTION, STATE}_2) \) and with initial state \( s_1 \), then the following clauses give a correct Event Calculus description of the system:

\[
\text{initiates}(E, \text{STATE}) \leftarrow \text{holds}_{at}(\text{OLDSTATE}, E), \\
\text{transition}(\text{OLDSTATE}, \text{ACTION, STATE}), \\
\text{act}(E, \text{ACTION}). \\
\text{terminates}(E, \text{STATE}) \leftarrow \text{holds}_{at}(\text{STATE}, E), \\
\text{transition}(\text{STATE, ACTION, NEWSTATE}), \\
\text{act}(E, \text{ACTION}), \\
\neg \text{STATE} = \text{NEWSTATE}.
\]

Initially \( (s_1) \).

These rules are a direct formalization of the intended meaning of a labeled transition system.

5 Discussion

Process algebras provide support for the representation of processes and synchronisation in the sense that these notions are hard-coded in the semantics of these languages. In contrast, open logic programming is a universal logic; it does not provide hard-coded support for representing dynamic systems, time and change, let alone the concepts of process and synchronisation. Event calculus explicates the laws of time and change, but still does not support the concepts of process and synchronisation. This observation might lead one to expect a high verbosity in our specification, due to the lack of support of the concepts. The verbosity in our specification is surprisingly low. Our specification contains about 30 main domain dependent clauses (not counting e.g. the initialisation clauses and the definition of \( \text{cnext} \)) with an average of 3-4 literals in the head. The specification in section 3 and the specification of the same protocol in a process algebra in [13], have about the same length. In general, our specification models directly and naturally the different states in which a process may be and the effects of actions. The control in the specification is mode-based: modes, represented by the \text{mode} attribute, are used to control the order of events. For example, a timer run-out brings the process in retransmitting mode during which it can only retransmit...
An alternative for this mode-based control would be to allow compound actions in Event Calculus. Such compound actions are described by pieces of imperative code and have been introduced already in [15] in the context of situation calculus. In the near future, we will investigate how to introduce such compound actions in event calculus.

Comparing our specification with a specification in a process algebra approach, we find an important difference on the level of the conceptualisation of the process concept. In a process algebra a process is a static, algebraic entity built up by combining simpler processes using pre-defined operators. In contrast, our Event Calculus specification models a process as a dynamic entity with an identity and an internal state. A process has attributes representing the internal state, it acts, has a restricted ability to sense actions executed by other entities and has the ability to execute actions which modify its internal state and a restricted part of the outer world.

The differences in view lead to extremely different specification styles. The Event Calculus specification models the state of the world at each moment: packets that are on the channel, contents of the slots in the sending window, current mode of the processes, expected frame and acknowledgement numbers, and so on. It also describes the evolution of the world as a result of events, and the preconditions of each event type. From this information, possible sequences of events (traces) can be derived.

Process algebra specifications do not model the evolving state of the world, but only the set of possible traces that can be derived from it. On the one hand, this is an advantage if the traces are the only thing one is interested in: a lot of unnecessary information is abstracted away. On the other hand, the loss of information limits the applicability of process algebra specifications to certain tasks.

Another consideration is that Event Calculus style specifications, due to the fact that they model the real-world parameters from which traces can be derived instead of the traces themselves, tend to be both easier to produce and easier to modify: preconditions and effects can be described for each event type independently, whereas calculating the possible traces requires taking into account all possible interactions. Moreover, changes in the effects of one event can have a considerable and complicated influence on the set of possible traces, where it only leads to the modification of one effect clause in an Event Calculus style specification.

So far, we have focused on the role of open logic programming for specification. Obviously, specifications are to be used in several ways, i.e. several kinds of reasoning on them are required.

For Lotos specifications, software tools have been developed for different computational tasks, including testing and simulation, verification and compilation of the specification in executable programs (for an overview see [2]). Similar tools can or are being developed for event calculus. In [19], a general purpose approach to simulation in event calculus is proposed. We plan to implement an interpreter for event calculus which uses an event calculus protocol specification to run the protocol specification. The system is similar to the simulation system but runs one process in the protocol and communicates with the other process in the distant location via an interface. A similar system has been developed for situation calculus in [15]. Different forms of verification can be done in event calculus and Open Logic Programming. For example, one requirement of the protocol specified above is that it should provide a perfect channel to the network layer, i.e. that frames sent by the network layer on one side arrive all exactly once and in the right order on the other side. We have actually proven this property of our specification using a second-order induction axiom on events. Another typical verification problem in the context of distributed systems is whether the protocol is
deadlock-free. A (simplified) condition which expresses that a deadlock arises at time $T$ is that no event can happen at time $T$, i.e. that for no action its preconditions are satisfied. The following formula $\Psi_d$ expresses this:

$$\exists T \forall A. \neg \text{precondition}(A, T)$$

A deadlock-free protocol should entail $\neg \Psi_d$. This is essentially a deductive problem. In case the specification does not entail this formula, a third type of problem arises: namely to explain why $\Psi_d$ can be true, i.e. how a deadlock may arise. This problem is a diagnosis problem: an abductive reasoner can be used to find a scenario (described in terms of the undefined predicates happens, $<$ and act) in which $\Psi_d$ is entailed. One may observe that this type of problem is formally equivalent to a planning problem in Event Calculus ([21]). In [8], we have shown how abductive extensions of SLDNF resolution, currently under development in the area of abductive logic programming, can be used for deduction, satisfiability checking and abduction in the context of open logic programming. We currently experimenting with the use of SLDNFA [6] for the above sorts of verification.

The general applicability of Event Calculus to a wide range of tasks involving dynamic systems allows for an integration of protocol specifications with other systems. For example, in this paper we have specified a communication protocol. This protocol will typically be used by processes exchanging information in a network. A related application is network management and diagnosis. Obviously, both network diagnosis and protocol specification require knowledge of low-level parameters of the network, the communications channels, the states of processes and the occurring events. For example, network diagnosis may need information on the frequency of packet losses on a particular channel, the average number of frames in unacknowledged slots, or the average number of retransmissions: information which can be obtained from our specification. If different special purpose languages are used to deal with protocols on the one hand (e.g. process algebra’s), and the network diagnosis on the other hand (e.g. Prolog rules), then integration of, and cooperation between these components becomes extremely difficult. It is a considerable advantage of Event Calculus that it provides one general description language for the entire system, which can be used as the basis specification and development for most (or all) of its components.

References


