[11] S. Gatziu and K. R. Dittrich. Detecting composite events in active databases using petri nets. In *Proceedings of the 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 2–9, February 1994.

[12] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th VLDB International Conference*, pages 327–338, 1992.

[13] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.

[14] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.

[15] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1977.

[16] I. Motakis and C. Zaniolo. Composite Temporal Events in Active Database Rules: A Logic-Oriented Approach. In *Proceedings of the 4th Intl. Conference on Deductive and Object-Oriented Databases*, 1995.

[17] I. Motakis and C. Zaniolo. Composite Temporal Events in Active Databases: A Formal Semantics. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, Workshops in Computing Series, pages 332–351. Springer, 1995.

[18] NASA, Lyndon Johnson Space Center, Software Technology Branch. *CLIPS 6.0 Reference Manual*, June 1993.

[19] N. W. Paton, J. Campin, A. A. Fernandes, and M. H. Williams. Semantics for Active Database Systems. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, pages 21–35, 1995.

[20] P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *Proceedings of the 14th Symposium on Principles of Database Systems*, pages 126–138, 1995.

[21] N. Pissinou, R. Snodgrass, R. Elmasri, I. Mumick, M. Ozsu, B. Pernici, A. Segev, B. Theodoulidis, and U. Dayal. Towards an infrastructure for temporal databases. *ACM-SIGMOD Record*, 23(1), March 1994.

[22] A. P. Sistla and O. Wolfson. Temporal Conditions and Integrity Constraints in Active Database Systems. In *Proc. of the 1995 ACM SIGMOD Intl. Conference on Management of Data*, pages 269–280, 1995.

[23] C. Zaniolo. A unified semantics for active and deductive databases. In *Proceedings of the 1st International Workshop on Rules in Database Systems*, pages 271–287, 1993.

constructs in such languages. This simplifies the task of specifying a language and comparing its differences to other languages.

In fact, the approach has been used in the paper to (i) provide a complete definition of the EPL active rule language, (ii) clarify the most salient semantic differences between languages such as ODE, Snoop and SAMOS, and (iii) expose some open issues in the definition of these languages, such as parameters, negation and simultaneous events.

Eventhough we have focused on formal semantics issues, a set of $Datalog_{1S}$ rules derived by our translation method can also be assigned a highly procedural forward chaining semantics for $Datalog\neg$, in the style of production systems [20]. By observing the form of the derived rules, we conclude that they can be evaluated in an incremental and *history-less* manner [6].

This paper leaves several issues for further research. One is that of formal characterization and comparison of the expressive power of different composite event specification languages, using concepts and tools that have been previously applied to Datalog-like languages [1]. The second is that of alternative evaluation strategies for composite events. In fact, while $Datalog_{1S}$ rules can be evaluated incrementally, as new events occur, they can also be evaluated and optimized as queries against an historical database. We are currently investigating these two problems.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] N. Arni, K. Ong, S. Tsur, and C. Zaniolo. LDL++: A second generation deductive database system. submitted for publication.

[3] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases. In A. Tansel et al., editor, *Temporal Databases: Theory, Design and Implementation*, chapter 13, pages 294–320. Benjamin/Cummings, 1993.

[4] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Anatomy of a composite event detector. Technical Report CIS TR-93-039, University of Florida, December 1993.

[5] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th VLDB Conference*, pages 606–617, September 1994.

[6] J. Chomicki. History-less checking of dynamic integrity constraints. In *Proceedings of the International Conference on Data Engineering*, pages 557–564, 1992.

[7] R. David. *Petri Nets and Grafcet: Tools for modeling discrete event systems*. Prentice Hall, New York, 1992.

[8] U. Dayal, E.N. Hanson, and J. Widom. Active Database Systems. In W. Kim, editor, *Modern Database Systems*. Addison Wesley, 1995.

[9] U. Dayal et al. The HiPAC Project: Combining active databases and timing constraints. *ACM-SIGMOD Record*, 17(1):51–70, March 1988.

[10] S. Gatziu and K. R. Dittrich. Events in an object-oriented database system. In *Proceedings of the First Intl. Conference on Rules in Database Systems*, pages 23–39, September 1993.

The *component by component* translation described above can produce a lengthy set of rules. Moreover, note that we have lost the direct correspondence between basic event occurrences and stages, which we have assumed in the previous sections.

We can solve these problems by providing a much simpler set of $Datalog_{1S}$ rules, that still models precisely the execution semantics of our example Petri Net and also maintains the one-to-one correspondence between external basic events and stages. This is achieved by folding the sequence of stages (states) in Table 3 into the one in Table 4.

In the new table, each external basic event creates only one new stage. This stage is defined by (a) the tokens residing at the regular places F and FF, at the end of the *sequence of firings* originated by the basic event, and (b) the token (if any) deposited into the output place E, during this sequence of firings.

Event	F	FF	E
	-	-	-
$G(y_a)$	-	-	-
$F(x_a)$	-	$(x_a,1)$	-
$F(x_b)$	$(x_b,2)$	$(x_a,1)$	-
$G(y_b)$	-	$(x_b,2)$	$(x_a y_b,1)$
$G(y_c)$	-	-	$(x_b y_c,2)$

Table 4: The simplified stage sequence for the execution sequence of Table 3.

The following $Datalog_{1S}$ rules defines the sequence of stages in Table 4:

$$\begin{array}{ll}
\texttt{in_F(X,N,s(J))} \leftarrow & \texttt{F(X,N,s(J)),\ in_FF(_,_,J)} \\
\texttt{in_F(X,N,s(J))} \leftarrow & \texttt{any(s(J)),\ in_F(X,N,J),\ } \neg\texttt{in_FF(X,N,s(J))} \\
\texttt{in_FF(X,N,s(J))} \leftarrow & \texttt{F(X,N,s(J)),\ } \neg\texttt{in_FF(_,_,J)} \\
\texttt{in_FF(X,N+1,s(J))} \leftarrow & \texttt{in_F(X,N+1,J),\ sat}_{\texttt{E}}\texttt{(_,_,N,s(J))} \\
\texttt{sat}_{\texttt{E}}\texttt{(X,Y,N,s(J))} \leftarrow & \texttt{G(Y,s(J)),\ in_FF(X,N,J)}
\end{array}$$

Note that no predicate is needed for auxiliary place H. Also, no predicate is needed for G, since an occurrence of G is either immediately matched with an occurrence of F, or it is "dumped". As a result, we obtain a much simpler set of rules that still defines precisely the execution semantics of our example Petri Net.

As in the Chronicle context example in the previous section, this set of rules is *XY–stratified.*

6 Conclusions

In this paper, we have proposed a general method for the definition of the semantics of composite event specification languages for active databases. Whereas different operational formalisms have been used in the past for different languages, we have shown that Datalog_{1S} provides a flexible and convenient declarative framework for defining the semantics of composite event

been signaled so far (and thus, the first N occurrences of F have been used).

In coloured ordinary Petri Nets there is not inherent measure of time, or model of time flow. Thus, ordinary Petri Nets make it possible to describe *what* happens, but not *when* it happens [7]. Aiming at resolving this issue, an extension of ordinary Petri Nets, called *synchronized* Petri Nets, has recently been proposed. They enable the modeling of systems whose evolutions are time dependent and where the succession of system *states* must be clearly and deterministically described. The *state* of a Petri Net is defined by the assignment of tokens to places.

In a synchronized Petri Net, gate firings are synchronized to external events and they obey a particular partial order defined by the structure of the Petri Net. An external event triggers a sequence of gate firings that occur in one or more *steps*, until the Petri Net reaches a *stable state*. This is exemplified in Table 3.

We now demonstrate how $Datalog_{1S}$ can be used to capture the execution semantics of a synchronized Petri Net. We assume that the occurrences of basic events are *non-simultaneous*, as required by Petri Nets (this is in fact a limitation of theirs). The following $Datalog_{1S}$ program defines the execution semantics of the Petri Net in Figure 4.

$$
\begin{array}{ll}
\texttt{in_F}(\texttt{X},\texttt{N},\texttt{J}) \leftarrow & \texttt{F}(\texttt{X},\texttt{N},\texttt{J}) \\
\texttt{in_F}(\texttt{X},\texttt{N},\texttt{s}(\texttt{J})) \leftarrow & \texttt{any}(\texttt{s}(\texttt{J})),\ \texttt{in_F}(\texttt{X},\texttt{N},\texttt{J}),\ \neg \texttt{t}_1(\texttt{X},\texttt{N},\texttt{s}(\texttt{J})) \\
 & \\
\texttt{in_H}(1,0) & \\
\texttt{in_H}(\texttt{N},\texttt{s}(\texttt{J})) \leftarrow & \texttt{any}(\texttt{s}(\texttt{J})),\ \texttt{in_H}(\texttt{N},\texttt{J}),\ \neg \texttt{t}_1(_,\texttt{N},\texttt{s}(\texttt{J})),\ \neg \texttt{t}_2(_,\texttt{N},\texttt{s}(\texttt{J})) \\
\texttt{in_H}(\texttt{N},\texttt{J}) \leftarrow & \texttt{t}_2(_,\texttt{N},\texttt{J}) \\
\texttt{in_H}(\texttt{N}+1,\texttt{J}) \leftarrow & \texttt{t}_3(_,_,\texttt{N},\texttt{J}) \\
 & \\
\texttt{in_G}(\texttt{Y},\texttt{J}) \leftarrow & \texttt{G}(\texttt{Y},\texttt{J}) \\
\texttt{in_G}(\texttt{Y},\texttt{s}(\texttt{J})) \leftarrow & \texttt{any}(\texttt{s}(\texttt{J})),\ \texttt{in_G}(\texttt{Y},\texttt{J}),\ \neg \texttt{t}_2(\texttt{Y},_,\texttt{s}(\texttt{J})),\ \neg \texttt{t}_3(_,\texttt{Y},_,\texttt{s}(\texttt{J})) \\
 & \\
\texttt{in_FF}(\texttt{X},\texttt{N},\texttt{J}) \leftarrow & \texttt{t}_1(\texttt{X},\texttt{N},\texttt{J}) \\
\texttt{in_FF}(\texttt{X},\texttt{N},\texttt{s}(\texttt{J})) \leftarrow & \texttt{any}(\texttt{s}(\texttt{J})),\ \texttt{in_F}(\texttt{X},\texttt{N},\texttt{J}),\ \neg \texttt{t}_3(\texttt{X},_,\texttt{N},\texttt{s}(\texttt{J})) \\
 & \\
\texttt{sat}_\texttt{E}(\texttt{X},\texttt{Y},\texttt{J}) \leftarrow & \texttt{t}_3(\texttt{X},\texttt{Y},_,\texttt{J}) \\
 & \\
\texttt{t}_1(\texttt{X},\texttt{N},\texttt{s}(\texttt{J})) \leftarrow & \texttt{in_F}(\texttt{X},\texttt{N},\texttt{J}),\ \texttt{in_H}(\texttt{N},\texttt{J}) \\
\texttt{t}_2(\texttt{Y},\texttt{N},\texttt{s}(\texttt{J})) \leftarrow & \texttt{in_G}(\texttt{Y},\texttt{J}),\ \texttt{in_H}(\texttt{N},\texttt{J}) \\
\texttt{t}_3(\texttt{X},\texttt{Y},\texttt{N},\texttt{s}(\texttt{J})) \leftarrow & \texttt{in_FF}(\texttt{X},\texttt{N},\texttt{J}),\ \texttt{in_G}(\texttt{Y},\texttt{J})
\end{array}
$$

An *order parameter* N for the occurrences of basic event F has been introduced. A different predicate is defined for each place and each gate.

Consider for instance the rules for place F. The first rule says that the token for the *N-th* occurrence of F is deposited into the place of F, at the stage of that occurrence. The second rule says that the token of *colour N* (corresponding to the *N-th* occurrence of F) is contained in place F at stage s(J), if it is contained there at stage J and gate t1 is not enabled at stage s(J), which would cause that token to be removed.

On the other hand, the rule for a gate expresses the fact that if all its inputs become active at stage J, then the gate is enabled at stage s(J). An output token is created from information passed from the inputs of gate t3 and it is deposited into the place for E — see the rules for t3 and $\texttt{sat}_\texttt{E}$.

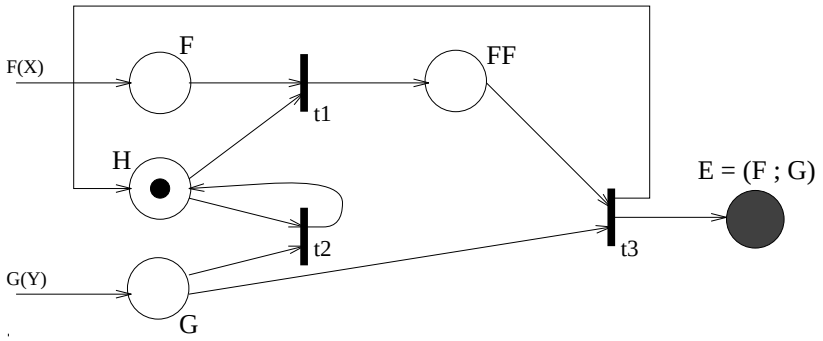

Figure 4: The Petri Net for the expression E = (F ; G).

where `F` contains the tokens for the rest of them. These different tokens are distinguished by their *colours*. Specifically, a token corresponding to an occurrence of `F` is *coloured* with an integer denoting the *order* of this occurrence among all the occurrences of `F` in the event history (this can be obtained by a method similar to that defining the *stage* of a basic event in the event history).

An occurrence of `G` is ignored, if there is not any token (occurrence) in `FF` to be matched with; observe how this is achieved by using the *auxiliary place* `H`. However, an occurrence `G(y)` that finds a token `F(x)` in `FF` is matched with it (this is achieved by firing the gate `t3`) and the occurrence of `E(x, y)` is then signaled. After that, `H` receives a token and becomes active again. Then, if there are any tokens in `F`, the token of the *oldest* unmatched occurrence of `F` is passed to `FF` through the firing of gate `t1`, so that it can be matched with the next occurrence of `G`. Note that this behaviour corresponds to the *chronicle* parameter context discussed in the previous section. The execution semantics of our Petri Net is best illustrated by an example, such as the one of Table 3.

Event	H	F	FF	G	E
	1	-	-	-	-
G(y_a)	1	-	-	y_a	-
t_2 fires	1	-	-	-	-
F(x_a)	1	(x_a,1)	-	-	-
t_1 fires	-	-	(x_a,1)	-	-
F(x_b)	-	(x_b,2)	(x_a,1)	-	-
G(y_b)	-	(x_b,2)	(x_a,1)	y_b	-
t_3 fires	2	(x_b,2)	-	-	(x_a,y_b)
t_1 fires	-	-	(x_b,2)	-	-
G(y_c)	-	-	(x_b,2)	y_c	-
t_3 fires	3	-	-	-	(x_b,y_c)

Table 3: An execution sequence of the Petri Net in Figure 3.

Since, places `F` and `FF` are *coloured*, the colours (order numbers) of their tokens are shown as well. Place `H` is also coloured and the colour of its token is `N+1`, if `N` occurrences of `E` have

of evaluation (prescribed by the XY–stratification of the program) leads to an efficient fixpoint, which is triggered by each basic event occurrence and terminates in a fixed number of steps.

- **Continuous Context.** Unlike the chronicle context, in the continuous context, a basic event may participate in more than one instance of the same composite event E. Multiple instances of E can thus occur simultaneously, at a stage J. All the basic events that participated in any of those instances are not considered after stage J, and therefore, they cannot participate in other instances of E.

  In our example, $(A(1), B(1))$ and $(A(2), B(1))$ are both detected in the continuous context. The following $Datalog_{1S}$ rules provide this semantics.

$$
\begin{array}{ll}
\mathtt{sat_1(X,J)} \leftarrow & \mathtt{A(X,J)} \\
\mathtt{has_sat_1(X,J)} \leftarrow & \mathtt{sat_1(X,J)} \\
\mathtt{has_sat_1(X,s(J))} \leftarrow & \mathtt{any(s(J)),\ has_sat_1(X,J),\ \neg sat_2(_,_,s(J))} \\
\mathtt{sat_2(X,Y,s(J))} \leftarrow & \mathtt{B(Y,s(J)),\ has_sat_1(X,J)}
\end{array}
$$

  The only difference between this set of rules and the one used for recent context is that in the continuous context, $\mathtt{sat_2}$ as opposed to $\mathtt{sat_1}$ is negated in the second rule for $\mathtt{has_sat_1}$. Also, this set of rules is *XY–stratified.*

- **Cumulative Context.** In this context, for each component event, all occurrences of the event are accumulated until the composite event is detected. Then, aggregation can be performed on these component event occurrences. Such aggregation operations have been implemented before in logic-based languages [2] and could be added to $Datalog_{1S}$ as well, but this is beyond the scope of this paper.

5.2 SAMOS and Petri Nets

The most distinguishing feature of SAMOS is its event detection mechanism, which is based on *coloured Petri Nets* [7]. A *coloured Petri Net* is an extended version of a classical Petri Net, that allows the flow of parameter bindings through it. In this way, the parameter passing within a composite event instance can be modeled. For a detailed description of how Petri Nets can be used for the detection of composite events, the reader is referred to [11]. Here we focus on the example of Figure 4 borrowed from [10], which illustrates the basic concepts. The Figure shows the initial state of the Petri Net, where a token is contained in the *auxiliary place* H.

This Petri Net accepts as input the occurrences of events F and G. Every such occurrence corresponds to a new token inserted into the Petri Net; this token contains the parameter bindings of the event occurrence. An output token is created when an occurrence F(x) is matched with an appropriate occurrence G(y). The output token containing the parameter bindings (x, y) is deposited into the output place E and a new occurrence of E is signaled. Right after that, this output token is removed from the Petri Net.

At a particular point in time, the tokens of the unmatched occurrences of F are contained in places F and FF. In particular, FF contains the token of the oldest among these occurrences,

- **Unrestricted Context**. In this parameter context, which has been assumed so far, all the instances of E are detected. These are:
  $(A(1), B(1))$, $(A(1), B(2))$, $(A(2), B(1))$ and $(A(2), B(2))$.

  The semantics of the Snoop SEQuence operator under the unrestricted context was explained earlier. There, an *auxiliary* predicate has_sat was used. To transform to different parameter contexts, we need to define other appropriate auxiliary predicates.

- **Recent Context.** In this context, only the most recent occurrences of the events (primitive or composite) are considered at each stage of history. The following instances of E are detected for example 8: $(A(2), B(1))$ and $(A(2), B(2))$.

  We can enforce this parameter context by defining an *auxiliary predicate* $\texttt{last_sat}_{\texttt{F}}(\texttt{X}, \texttt{J})$ for each Snoop subexpression F(X), where for a particular stage J, X denotes the parameter bindings vector of the *last* occurrence of F, *before or at* this stage. The following stratified $Datalog_{1S}$ rules define the semantics of our example Snoop expression, in the recent context:

$$
\begin{array}{l}
\texttt{sat}_1(\texttt{X}, \texttt{J}) \leftarrow \quad \texttt{A}(\texttt{X}, \texttt{J}) \\
\texttt{last_sat}_1(\texttt{X}, \texttt{J}) \leftarrow \quad \texttt{sat}_1(\texttt{X}, \texttt{J}) \\
\texttt{last_sat}_1(\texttt{X}, \texttt{s(J)}) \leftarrow \texttt{any(s(J))},\ \texttt{last_sat}_1(\texttt{X}, \texttt{J}),\ \neg\texttt{sat}_1(_, \texttt{s(J)}) \\
\\
\texttt{sat}_2(\texttt{X}, \texttt{Y}, \texttt{s(J)}) \leftarrow \quad \texttt{B}(\texttt{Y}, \texttt{s(J)}),\ \texttt{last_sat}_1(\texttt{X}, \texttt{J})
\end{array}
$$

- **Chronicle Context**. In the chronicle context, when a composite event E is satisfied, its parameter bindings are obtained from the oldest *unused* occurrences of its component events, that satisfy the precedence requirements of E (*unused* implying that the same basic event occurrence can participate in *at most one* instance of a particular composite event).

  The two instances of our example event that would be detected in the chronicle context are: $(A(1), B(1))$ and $(A(2), B(2))$. The following $Datalog_{1S}$ program provides this semantics. Observe that as in Section 4.7, we show here the satisfaction time arguments of the satisfaction predicates.

$$
\begin{array}{l}
\texttt{sat}_1(\texttt{X}, \texttt{X.Time}, \texttt{J}) \leftarrow \quad \texttt{A}(\texttt{X}, \texttt{J}) \\
\texttt{unused}_1(\texttt{X}, \texttt{T}, \texttt{J}) \leftarrow \quad \texttt{sat}_1(\texttt{X}, \texttt{T}, \texttt{J}) \\
\texttt{unused}_1(\texttt{X}, \texttt{T}, \texttt{s(J)}) \leftarrow \quad \texttt{any}(_, \texttt{s(J)}),\ \texttt{unused}_1(\texttt{X}, \texttt{T}, \texttt{J}),\ \neg\texttt{sat}_2(\texttt{X}, _, _, \texttt{s(J)}) \\
\\
\texttt{not_oldest}_1(\texttt{T1}, \texttt{J}) \leftarrow \quad \texttt{unused}_1(_, \texttt{T2}, \texttt{J}),\ \texttt{T2} < \texttt{T1} \\
\texttt{sat}_2(\texttt{X}, \texttt{Y}, \texttt{Y.Time}, \texttt{s(J)}) \leftarrow \texttt{B}(\texttt{Y}, \texttt{s(J)}),\ \texttt{unused}_1(\texttt{X}, \texttt{T1}, \texttt{J}),\ \neg\texttt{not_oldest}_1(\texttt{T1}, \texttt{J})
\end{array}
$$

  Eventhough these rules are not stratified, they are locally stratified and *XY–stratified* [23]; therefore they can be evaluated in an incremental and efficient manner. To see that, observe that the computation of the contents of $\texttt{unused}_1$ and $\texttt{not_oldest}_1$ at stage J can be followed by the computation of the contents of $\texttt{sat}_2$ at stage s(J), which can then be followed by the computation of the contents of $\texttt{unused}_1$ at stage s(J). This order

occurred. This is different from EPL, where the variables inside a disjunct are not exported outside it.

3. **Aperiodic Operator A.** The event $\texttt{E} = \texttt{A}(\texttt{E}_1, \texttt{E}_2, \texttt{E}_3)$ occurs each time $\texttt{E}_2$ occurs, within the *closed* time interval started by an occurrence of $\texttt{E}_1$ and ended by the *first subsequent* occurrence of $\texttt{E}_3$. Its parameters consist of the parameters of $\texttt{E}_2$. Its translation into $Datalog_{1S}$ rules follows:

$$\begin{array}{ll} \texttt{between}_{1,3}(\texttt{J}) \leftarrow & \texttt{sat}_1(_, \texttt{J}) \\ \texttt{between}_{1,3}(\texttt{s}(\texttt{J})) \leftarrow & \texttt{any}(\texttt{s}(\texttt{J})),\ \texttt{between}_{1,3}(\texttt{J}),\ \neg\texttt{sat}_3(_, \texttt{J}) \\ \texttt{sat}_{\texttt{E}}(\texttt{X}, \texttt{J}) \leftarrow & \texttt{between}_{1,3}(\texttt{J}),\ \texttt{sat}_2(\texttt{X}, \texttt{J}) \end{array}$$

The auxiliary predicate $\texttt{between}_{1,3}$, is *true* at all stages in the closed time interval between an occurrence of $\texttt{E}_1$ and the *first subsequent* occurrence of $\texttt{E}_3$. X denotes the parameters of $\texttt{E}_2$.

For an outline of how to handle the aperiodic cumulative operator A^* (used mainly for computing aggregates), we refer to the discussion of the cumulative parameter context.

4. **NOT.** The event $\texttt{E} = \neg(\texttt{E}_2)[\texttt{E}_1, \texttt{E}_3]$ detects the non-occurrence of $\texttt{E}_2$ in the *closed* interval between an occurrence of $\texttt{E}_1$ and its *first subsequent* occurrence of $\texttt{E}_3$. It is signaled at that occurrence of $\texttt{E}_3$. Formally:

$$\begin{array}{ll} \texttt{between}_{1,23}(\texttt{J}) \leftarrow & \texttt{sat}_1(_, \texttt{J}) \\ \texttt{between}_{1,23}(\texttt{s}(\texttt{J})) \leftarrow & \texttt{any}(\texttt{s}(\texttt{J})),\ \texttt{between}_{1,23}(\texttt{J}),\ \neg\texttt{sat}_2(_, \texttt{J}),\ \neg\texttt{sat}_3(_, \texttt{J}) \\ \texttt{sat}_{\texttt{E}}(\texttt{J}) \leftarrow & \texttt{between}_{13}(\texttt{J}),\ \texttt{sat}_3(_, \texttt{J}),\ \neg\texttt{sat}_2(_, \texttt{J}) \end{array}$$

The auxiliary predicate $\texttt{between}_{1,23}$ is *true* at all stages in the closed time interval between an occurrence of $\texttt{E}_1$ and the *first subsequent* occurrence of *either* $\texttt{E}_2$ *or* $\texttt{E}_3$.

The AND and ANY constructs of Snoop have been implemented separately, for efficiency reasons. However, they can be expressed in terms of the SEQ and the OR constructs.

Because of space limitations, we do not discuss the periodic event operators P and P^*, since the semantics of explicit time events in Snoop are different from the semantics defined in Section 4.7.

5.1.2 Parameter Contexts

A novelty offered by Snoop is the *parameters contexts*. As it is explained in [5], parameter contexts provide the ability of precisely matching the application needs for different interpretations of temporal sequences, arising from multiple occurrences of the event types participating in these sequences.

Thus, we need to extend our translation procedure accordingly.

Example 8 *The Snoop expression* `E(X,Y) = (A(X),B(Y))`*, and the event history:* $\ldots, A(1), \ldots, A(2), \ldots, B(1), \ldots, B(2)$, *where* A *and* B *denote basic events with parameters* X *and* Y *respectively.*

use the standard translation rule, and the stage value of $\mathtt{sat}_3$ must be $\mathtt{s(J)}$, with $\mathtt{J}$ being the satisfaction stage of $\mathtt{sat}_2$, which in turn is equal to that of $\mathtt{sat}_1$, as it should.

Observe that this produces a semantically correct translation, even in the case where clock events follow each other, without any intervening basic event. The effect is that if $\mathtt{J}$ and $\mathtt{s(J)}$ are two consecutive stages in the module's event history, with timestamps $\mathtt{T}_1$ and $\mathtt{T}_2$ respectively, all clock events $\mathtt{clock(T)}$, such that $\mathtt{T}_1 \leq \mathtt{T} < \mathtt{T}_2$, occur at stage $\mathtt{J}$.

The formal semantics of clock events appearing in relaxed sequences, or other composite events can be realized taking into account the above basic notions. We mention however, that clock events or (sub)sequence expressions ending in clock events cannot be negated, since this would result in an unclear semantics.

5 Application to Other Systems

One of the most appealing characteristics of the proposed method is its generality, whereby it can be used for the formal definition of the concepts appearing in previous systems.

For the case of ODE, this is straightforward, given the many similarities between the EPL and ODE constructs. The most important difference is that, in ODE, the *relaxed sequence* is a basic construct and the *immediate sequence* is a derivative one. Thus, the semantics of ODE can be defined by a syntax-directed translation from ODE expressions into $Datalog_{1S}$ rules, similar to that used for EPL. In fact, the application of the technique to ODE clarifies the definition of the disjunction, which seems to be problematic in [12]. There, only conjunction is defined as a primitive construct, while disjunction is defined through negation, via De Morgan's law. This is a problem in the presence of variables and also when disjunctions with multiple-stage durations are concerned (a negated event has always single stage duration).

Such translation procedures can also be defined for Snoop and SAMOS's language, that have a different flavour from EPL and ODE. One of the differences is that in Snoop and SAMOS, the meaning of the sequence of two events $\mathtt{E}_1$ and $\mathtt{E}_2$ is equivalent to the meaning of $\mathtt{prior(E_1, E_2)}$ in EPL and ODE.

In the following section, we define the semantics of the main Snoop constructs and we explain how its concept of *parameter contexts* can be incorporated in our method. Then, for SAMOS, we give a very detailed example of how we can formally define the execution semantics of Petri Nets, which have been used for its implementation.

5.1 SNOOP

5.1.1 The Snoop Constructs

1. **SEQ (;).** Since the event $\mathtt{E} = (\mathtt{E_1(X)};\ \mathtt{E_2(Y)})$ is equivalent to $\mathtt{prior(E_1(X),\ E_2(Y))}$ in EPL and ODE, the rule for $\mathtt{sat_E}$ is defined as it is explained in sections 4.4 and 4.5:

$$\mathtt{sat_E(X, Y, s(J))} \leftarrow\ \mathtt{sat_{E_2}(Y, s(J)),\ has_sat_{E_1}(X, J)}$$

2. **OR.** This is similar to EPL's disjunction, but the scope rules are different. In Snoop, the parameters of a disjunction's instance are the parameters of the disjunct that actually

$$
\begin{array}{ll}
\mathtt{sat_1(X,J)} \leftarrow & \mathtt{upd_A(X,J)}. \\
\mathtt{sat_2(X,Y,s(J))} \leftarrow & \mathtt{upd_B(Y,s(J))},\ \mathtt{sat_1(X,J)},\ \mathtt{q_b(X,Y)}. \\
\mathtt{sat_3(X,Z,s(J))} \leftarrow & \mathtt{ins_C(Z,s(J))},\ \mathtt{sat_1(X,J)},\ \mathtt{q_c(X,Z)}. \\
\mathtt{sat_4(X,J)} \leftarrow & \mathtt{sat_2(X,Y,J)}. \\
\mathtt{sat_4(X,J)} \leftarrow & \mathtt{sat_3(X,Z,J)}. \\
\mathtt{sat_5(X,s(J))} \leftarrow & \mathtt{any(s(J))},\ \mathtt{sat_1(X,J)},\ \neg\mathtt{sat_4(_,s(J))}. \\
\mathtt{sat_E(X,J)} \leftarrow & \mathtt{sat_5(X,J)}.
\end{array}
$$

The rule for $\mathtt{E_5}$ expresses the fact that $\mathtt{E_5}$ occurs at the stage immediately following $\mathtt{E_1}$'s occurrence, if neither $\mathtt{E_2}$, nor $\mathtt{E_3}$ occur at this stage.

$\mathtt{E_1}$ is considered to be a possible predecessor of $\mathtt{E_2}$ and $\mathtt{E_3}$ as well. Generally, in a sequence expression of the form `(F,!G)`, the subexpression `G` is evaluated with respect to the basic event history starting right after the satisfaction of `F`. Using Table 1, we get for instance:
$\mathtt{PPS(2)} = \mathtt{PPS(4)} = \mathtt{PPS(5)} = \{1\}$

4.7 Clock Events

Clock events (explicit time events) are treated differently from basic events. Take for instance the following EPL event pattern:

$$\mathtt{(\ upd(ACC(X)),\ clock(1428),\ ins(ACC(Y))\ )}$$

In order to handle clock events, we need to introduce a *satisfaction time* attribute, into each event's satisfaction predicate. Thus, the satisfaction predicate of general event `E` has now the form $\mathtt{sat_E(X,T,J)}$. The satisfaction time for a basic event is its timestamp, where for a clock event, that is the value of its temporal expression. The rules in Table 2 can be augmented accordingly, so that the satisfaction times of composite events are defined. [3]

In this simple example, `X.Time` and `Y.Time` are the satisfaction times of $\mathtt{sat_1}$ and $\mathtt{sat_3}$, which denote the satisfaction predicates of the two basic events, according to the usual translation. Thus, the following rules define its semantics:

$$
\begin{array}{ll}
\mathtt{sat_1(X,X.Time,J)} \leftarrow & \mathtt{upd_ACC(X,J)} \\
\mathtt{sat_2(X,1428,J)} \leftarrow & \mathtt{sat_1(X,T,J)},\ \mathtt{T} \leq 1428,\ \neg\mathtt{between(T,1428)} \\
\mathtt{sat_3(X,Y,Y.Time,s(J))} \leftarrow & \mathtt{ins_ACC(Y,s(J))},\ \mathtt{sat_2(X,_,J)}
\end{array}
$$

The basic approach is to treat `clock(Time-Expr)` (where `Time-Expr` evaluates to a time constant), as a constraint. What is required to enforce the immediate sequence semantics, is that no *basic* event occurs between the satisfaction of the immediate predecessor of the clock event and time `Time-Expr`.

The satisfaction stage of $\mathtt{sat_2}$ remains the same as the satisfaction stage of its immediate predecessor $\mathtt{sat_1}$, insofar as this corresponds to a clock event. For $\mathtt{sat_3}$ on the other hand, we

[3] In order to keep the presentation easier to follow, we have avoided showing the *satisfaction time* attributes, except when it is necessary. Such a case is the *clock events*, and another case appears in the next section, where we discuss the *chronicle parameter context.*

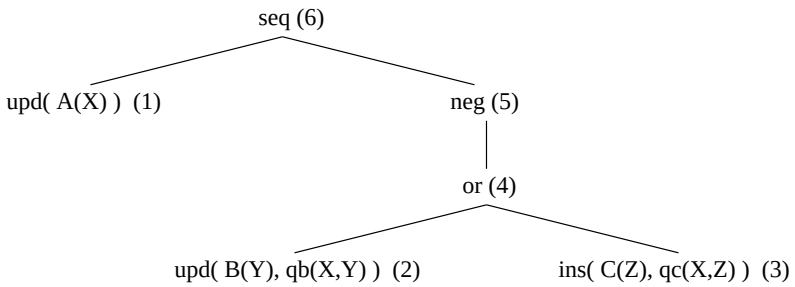

Figure 3: An EPL expression with a negated subevent

4.6 Negation

Handling negation of arbitrary composite events has been problematic in most of the previous approaches, which therefore support only limited forms of negation.

Using $Datalog_{1S}$, the semantics of general negation can be easily defined. For instance, for the negated qualified basic event
`E = !ins(ACC(X), X.Type = "Savings")`, we have (using domain variables):

$$\begin{aligned} \mathtt{sat_F(Accno, Owner, Type, Bal, Time, J)} \leftarrow & \\ & \mathtt{ins_ACC(Accno, Owner, Type, Bal, Time, J),\ Type} = \text{“Savings”} \\ \mathtt{sat_E(J)} \leftarrow & \quad \mathtt{any(J),\ \neg sat_F(_, _, _, _, _, J).} \end{aligned}$$

The second rule expresses the fact that `E` occurs at every stage where *some* basic event occurs, but `F` does not occur. Referring to the `hist_monit` table, `E` occurs at every stage, except stage 3, where an insertion of a savings account is recorded.

As these rules show, the variables defined inside a negated event are not exported outside it. This restriction ensures the safety and domain-independence of EPL expressions.

The general case is similar. The second rule above can still be used for the negated event `E = !F`, where `F` is an arbitrary event. Note that at every stage that $\mathtt{sat_E}$ is satisfied, we have the occurrence of a different instance of `E`, and thus, every such instance has *single stage duration*.

The following example illustrates how negated composite events are handled. This example also demonstrates a disjunctive event.

Example 7 *The EPL expression for Figure 3.*
$(\ \mathtt{upd(A(X))},\quad !\ \{\mathtt{upd(B(Y), q_b(X,Y))},\ \mathtt{ins(C(Z), q_c(X,Z))}\}\ )$

The satisfaction of the negated event $\mathtt{E_5}$ (at a stage where neither $\mathtt{E_2}$, nor $\mathtt{E_3}$ occur) must follow the occurrence of basic event $\mathtt{E_1}$.

Since a negated event instance has *single stage duration*, negated events are treated similarly to basic events, as far as *ordering* is concerned. Namely, one rule of the form shown in Table 2 is created for each of the possible predecessor of a negated event. In this example, the only possible predecessor of the negated event $\mathtt{E_5}$ is $\mathtt{E_1}$.

The $Datalog_{1S}$ rules for Example 7 follow:

can be translated into the following rules:

$$\begin{array}{ll}
\texttt{sat}_1(\texttt{X},\texttt{J}) \leftarrow & \texttt{upd_ACC}(\texttt{X},\texttt{J}),\ \texttt{q}_1(\texttt{X}) \\
\texttt{has_sat}_1(\texttt{X},\texttt{J}) \leftarrow & \texttt{sat}_1(\texttt{X},\texttt{J}) \\
\texttt{has_sat}_1(\texttt{X},\texttt{s}(\texttt{J})) \leftarrow & \texttt{any}(\texttt{s}(\texttt{J})),\ \texttt{has_sat}_1(\texttt{X},\texttt{J}) \\
\texttt{sat}_2(\texttt{X},\texttt{Y},\texttt{s}(\texttt{J})) \leftarrow & \texttt{upd_ACC}(\texttt{Y},\texttt{s}(\texttt{J})),\ \texttt{has_sat}_1(\texttt{X},\texttt{J}),\ \texttt{q}_2(\texttt{X},\texttt{Y})
\end{array}$$

For an event type E, the auxiliary predicate $\texttt{has_sat}_\texttt{E}$ contains at a particular stage J, one tuple for each occurrence of E before or at this stage.

4.5 Conjunction and Simultaneous Events

A *conjunctive event* E = (F & G) occurs at a stage where both F and G occur. The instances of F and G that cause E to be satisfied may have different starting stages. F and G are *evaluated independently of each other* (in parallel).

Using the conjunction construct, we can express sequences based on event occurrences, as opposed to event instances that follow each other. An example is the definition of prior, which is repeated here (variables are included):

$$\texttt{E}(\texttt{X},\texttt{Y}) = \texttt{prior}(\texttt{F}(\texttt{X}),\texttt{G}(\texttt{Y})) = (\ [\texttt{F}(\texttt{X}),\texttt{any}]\ \&\ \texttt{G}(\texttt{Y})\ )$$

Assuming that the rules for F(X) and G(Y) have been generated and that an auxiliary predicate $\texttt{has_sat}_\texttt{F}$ has been defined as explained in the previous section, the satisfaction predicate of E is defined as:

$$\texttt{sat}_\texttt{E}(\texttt{X},\texttt{Y},\texttt{s}(\texttt{J})) \leftarrow\ \texttt{sat}_\texttt{G}(\texttt{Y},\texttt{s}(\texttt{J})),\ \texttt{has_sat}_\texttt{F}(\texttt{X},\texttt{J})$$

Observe that the number of rules generated for a conjunctive event equals the sum of the numbers of rules generated for its two conjuncts, plus one extra rule. Thus, we do not have an exponential blow-up problem.

Conjunction can also be used to handle *simultaneous* events. Consider e.g.

$$\texttt{E} = (\ \texttt{upd}(\texttt{A}(\texttt{X})),\ (\ \texttt{ins}(\texttt{B}(\texttt{Y}))\ \&\ \texttt{del}(\texttt{C}(\texttt{Z}))\ )\ ),$$

This composite event occurs when the first basic event is immediately followed by the simultaneous occurrence of the last two basic events. Its translation follows:

$$\begin{array}{ll}
\texttt{sat}_1(\texttt{X},\texttt{J}) \leftarrow & \texttt{upd_A}(\texttt{X},\texttt{J}) \\
\texttt{sat}_2(\texttt{X},\texttt{Y},\texttt{J}) \leftarrow & \texttt{ins_B}(\texttt{Y},\texttt{s}(\texttt{J})),\ \texttt{sat}_1(\texttt{X},\texttt{J}) \\
\texttt{sat}_3(\texttt{X},\texttt{Z},\texttt{J}) \leftarrow & \texttt{del_C}(\texttt{Z},\texttt{s}(\texttt{J})),\ \texttt{sat}_1(\texttt{X},\texttt{J}) \\
\texttt{sat}_\texttt{E}(\texttt{X},\texttt{Y},\texttt{Z},\texttt{J}) \leftarrow & \texttt{sat}_2(\texttt{X},\texttt{Y},\texttt{J}),\ \texttt{sat}_3(\texttt{X},\texttt{Z},\texttt{J})
\end{array}$$

Eventhough simultaneous events have not been discussed in previous approaches, there are many cases where this functionality is desired. As mentioned in the beginning of section 4, this is necessary when *transaction boundaries granularity* must be modeled. Simultaneous events may also occur in a distributed or multiprocessor environment.

$PPS(2) = \{2\} \cup PPS(4) = \{1,2,4\}$
$PPS(3) = \{2\} \cup PPS(4) = \{1,2,4\}$

The variable scopes for this example have been visualized in Figure 2 using contours. Basic events are listed in order of their appearance in the EPL expression and all basic events in the same *star* subexpression are enclosed within the same contour. The condition of a basic event E can refer to all variables whose scopes extends to this event. Using this information and the *PPS* sets of the basic events, the following $Datalog_{1S}$ rules are derived for Example 6:

$$
\begin{array}{ll}
\mathtt{sat_1(X,J)} \leftarrow & \mathtt{upd_A(X,J),\ q_a(X).} \\
\mathtt{sat_2(X,Y,s(J))} \leftarrow & \mathtt{ins_B(Y,s(J)),\ sat_1(X,J),\ q_b(X,Y).} \\
\mathtt{sat_2(X,Y,s(J))} \leftarrow & \mathtt{ins_B(Y,s(J)),\ sat_2(X,_,J),\ q_b(X,Y).} \\
\mathtt{sat_2(X,Y,s(J))} \leftarrow & \mathtt{ins_B(Y,s(J)),\ sat_4(X,_,J),\ q_b(X,Y).} \\
\mathtt{sat_3(X,Z,s(J))} \leftarrow & \mathtt{del_C(Z,s(J)),\ sat_1(X,J),\ q_c(X,Z).} \\
\mathtt{sat_3(X,Z,s(J))} \leftarrow & \mathtt{del_C(Z,s(J)),\ sat_2(X,_,J),\ q_c(X,Z).} \\
\mathtt{sat_3(X,Z,s(J))} \leftarrow & \mathtt{del_C(Z,s(J)),\ sat_4(X,_,J),\ q_c(X,Z).} \\
\mathtt{sat_4(X,Z,J)} \leftarrow & \mathtt{sat_3(X,Z,J).} \\
\mathtt{sat_5(X,V,s(J))} \leftarrow & \mathtt{upd_D(V,s(J)),\ sat_1(X,J),\ q_d(X,V).} \\
\mathtt{sat_5(X,V,s(J))} \leftarrow & \mathtt{upd_D(V,s(J)),\ sat_4(X,_,J),\ q_d(X,V).} \\
\mathtt{sat_6(X,V,J)} \leftarrow & \mathtt{sat_5(X,V,J).} \\
\mathtt{sat_E(X,V,J)} \leftarrow & \mathtt{sat_6(X,V,J).}
\end{array}
$$

Consider for instance $\mathtt{E_2} = \mathtt{ins(B(Y), q_b(X,Y))}$. This basic event may immediately follow an occurrence of basic event $\mathtt{E_1}$, or another occurrence of $\mathtt{E_2}$ (because of the innermost star), or an occurrence of a star subsequence $\mathtt{E_4}$ (because of the outermost * iteration).

The satisfaction predicates for the `seq` and `*seq` nodes are defined through *copy-rules*. These predicates are not needed, unless such a node is a possible predecessor of some basic event, as is the case of $\mathtt{E_4}$. However, we have included them in our presentation for clarity reasons.

As demonstrated in this example, EPL scope rules are implemented, by passing variables through the satisfaction predicates, to the conditions of all the basic events within the scope of the variables.

4.4 Relaxed Sequences

In section 3.2, EPL derivative constructs such as `any`, `prior`, and `relaxed sequences` were defined in terms of the basic constructs. Thus, a translation into $Datalog_{1S}$ need not be given explicitly. Yet, a direct translation is often desirable, as it leads to much more efficient implementation. For instance, `any` need not be defined as the disjunction of all basic events in the module of interest, but can be simply derived as it is shown in the first row of Table 1.

A relaxed sequence is treated similarly to an immediate sequence; e.g. the rules of Table 1 for an immediate sequence remain intact in the case of a relaxed sequence. The only difference is that in [F, G], an instance of G may start at some stage later, but not necessarily immediately after an occurrence of F. By using an *auxiliary predicate* $\mathtt{has_sat_1}$, the relaxed sequence

$$\mathtt{E} = [\ \mathtt{upd(ACC(X), q_1(X)),\ upd(ACC(Y), q_2(X,Y))}\ ],$$

Event Type E	*Datalog$_{1S}$ Rule Templates*
Any Basic Event	$any(J) \leftarrow hist_monit(_, _, _, J)$
Qual. Basic Event *evt(R(X), Cond)*	*for each* $P \in PPS(E)$ $sat_E(IV, X, s(J)) \leftarrow evt_R(X, s(J)),\ sat_P(IV, _, J),$ $Cond(IV, X)$
seq(F(X),G(Y))	$sat_E(IV, X, Y, J) \leftarrow sat_G(IV, X, Y, J)$
$*$*seq(F(X),G(Y))*	$sat_E(IV, Y, J) \leftarrow sat_G(IV, Y, J)$
or(F(X),G(Y))	$sat_E(IV, J) \leftarrow sat_F(IV, X, J)$ $sat_E(IV, J) \leftarrow sat_G(IV, Y, J)$
and(F(X),G(Y))	$sat_E(X, Y, IV, J) \leftarrow sat_F(IV, X, J),\ sat_G(IV, Y, J)$
neg(F(X))	*for each* $P \in PPS(E)$ $sat_E(IV, s(J)) \leftarrow any(s(J)),\ sat_P(IV, _, J),$ $\neg sat_F(_, s(J))$
ε	*for each* $P \in PPS(E)$, $sat_E(IV, J) \leftarrow sat_P(IV, _, J)$

Table 2: *Datalog$_{1S}$* rule templates for the basic constructs of EPL.

the variables defined in its conjuncts are exported, where in a disjunction, none of the variables defined in its disjuncts is exported. The rule for *negated events* is explained in Section 4.6.

Note also, that the variables of a satisfaction predicate consists of the union of its exported variables, plus the variables imported into it.

4.3 Immediate and Star Sequences

Having illustrated how immediate sequences are handled, we move on to the case of star sequences, which is somewhat more complicated.

Consider e.g., the EPL expression `E = (F, G, *:H, K)`. Obviously, `PPS(G) = {F}`. However, because of the star operator, an instance of `H` might immediately follow either an occurrence of `G`, or a previous occurrence of `H`. Therefore, `PPS(H) = {G,H}`. Similarly, an instance of `K` may immediately follow either an occurrence of `G` (zero instances of `H` after `G`), or the last occurrence of `H` and thus, `PPS(K) = {G,H}`. Variables defined in a *star* subexpression are not exported to subexpressions that follow. The fourth row of Table 1 provides the formal details.

Example 6 shows a more complicated case, where *star* subexpressions are nested. Referring to Figure 2 and using Table 1, we get:

$PPS(7) = \emptyset$
$PPS(1) = PPS(7) = \emptyset$
$PPS(6) = \{1\}$
$PPS(4) = \{4\} \cup PPS(6) = \{1, 4\}$
$PPS(5) = \{4\} \cup PPS(6) = \{1, 4\}$

EvtType E	*PPS*	*EVar(E)*	*IVar*
evt(R(X))	—	X	—
seq(F,G)	$PPS(F) = PPS(E)$ $PPS(G) = F$	$EVar(F) \cup EVar(G)$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E) \cup EVar(F)$
**seq(F,G)*	$PPS(F) = F \cup PPS(E)$ $PPS(G) = F \cup PPS(E)$	$EVar(G)$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E)$
or(F,G)	$PPS(F) = PPS(E)$ $PPS(G) = PPS(E)$	$\emptyset$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E)$
and(F,G)	$PPS(F) = PPS(E)$ $PPS(G) = PPS(E)$	$EVar(F) \cup EVar(G)$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E)$
neg(F)	$PPS(F) = PPS(E)$	$\emptyset$	$IVar(F) = IVar(E)$

Table 1: An attribute grammar for syntax-directed translation from EPL to $Datalog_{1S}$.

instance of `Q` (i.e., the instance of `Q` can begin at the next stage). Because of disjunctions and the star operator, a particular subevent may have many possible predecessors.

For example, consider the immediate sequence event: `E = seq(F,G)`. `F` is the only possible predecessor of `G`; but the set of possible predecessors of `F` depends on which events may precede `E`—i.e., `F` inherits `E`'s possible predecessors.

The remaining two columns of Table 1 describe the scope rules for variables in EPL. The third column shows the set of *exported variables* of an EPL expression. These are variables defined in the expression (variables appearing in basic events within this expression), whose scopes extends past the satisfaction of the expression. The fourth column contains for each subevent `Q` of an EPL expression, the set of variables *imported* into `Q` (variables defined outside `Q`, whose scopes extends to `Q`).

Again, for `E = seq(F,G)`, the set of variables exported from `E` is the union of the variables exported from `F` and `G`. On the other hand, `E` might have imported some variable names from previous events and if so, these are also passed down to `F` and `G`. In addition to variables inherited by `E`, variables imported into `G` include those exported from `F`.

Using the information of Table 1, the generation of the actual rules is simple, as shown in Table 2. Observe that except for basic events, `X` and `Y` denote *sets* of exported variables defined in various subevents, where `IV` denotes the *set* of imported variables into a particular event type `E`. The anonymous variable `_` has replaced all variables that must be kept local.

The first row in this table defines directly the event `any` (see Section 2.2), using a module's event history relation. This event will be used in several of the following examples.

The second row deals with qualified basic events having some possible predecessors (the case of a basic event with no possible predecessors is trivial). Such an event `E` is satisfied at stage `s(J)`, when: (1) a possible predecessor of E was satisfied at stage `J`, (2) `E` occurs at stage `s(J)`, and (3) The condition of `E` is satisfied. Example 5 illustrates this translation.

The rules for disjunction and conjunction are apparent. Observe that in a conjunction, all

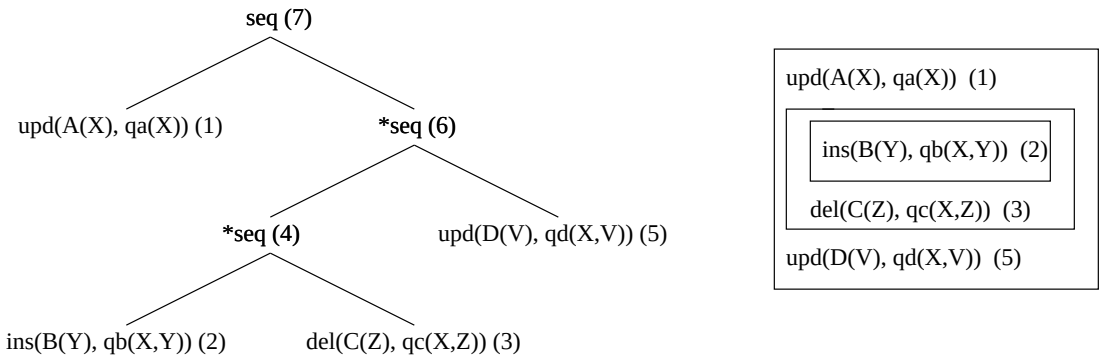

Figure 2: An EPL expression with Immediate and Star Sequences

1. $seq(E_i, E_j) \equiv (E_i, E_j)$.[1]
2. $*seq(E_i, E_j) \equiv (*E_i, E_j)$.[2]
3. $and(E_i, E_j) \equiv E_i \ \& \ E_j$.
4. $or(E_i, E_j) \equiv \{E_i, E_j\}$.
5. $neg(E) \equiv !E$.

Example 6 *The EPL expression*
$(\mathtt{upd(A(X), q_a(X))},\ *:(*:\mathtt{ins(B(Y), q_b(X,Y))},\ \mathtt{del(C(Z), q_c(X,Z))}),\ \mathtt{upd(D(V), q_d(X,V))})$

The parse tree for the expression of Example 6 is shown in Figure 2. The nodes of the tree are numbered according to the postorder traversal sequence.

Each node `i` corresponds to a subevent $\mathtt{E_i}$, and the *satisfaction predicate* of $\mathtt{E_i}$ is denoted as $\mathtt{sat_i}$. For a subevent expression, its *satisfaction predicate* contains one tuple for each distinct (in terms of variable bindings and stage) occurrence of this subevent.

4.2 The Translation Procedure

As demonstrated by the last translation example, for a composite EPL expression, the $Datalog_{1S}$ rules must model (i) the transmission of variable bindings according to the scope rules of the various constructs, so that variables can be matched and conditions can be checked, and (ii) the temporal precedences among the various subevents.

Table 1 describes how this information is derived for each basic EPL construct (formally it defines a simple attribute grammar for syntax-directed translation).

For each subevent `Q` of an EPL event `E`, the second column in Table 1 defines the *Possible Predecessors Set* of `Q`, denoted as `PPS(Q)`. A subevent `P` is a possible predecessor of `Q` within `E`, if in an instance of `E`, the satisfaction of `P` can *immediately* precede the first basic event of an

[1] $(E_1, E_2, \ldots, E_{n-1}, E_n) = seq(E_1, seq(E_2, \ldots, seq(E_{n-1}, E_n)\ldots))$. Similarly for the relaxed sequence, the conjunction and the disjunction constructs.

[2] We use the binary construct `*seq` in place of the `*:` EPL construct, so that the representation is more compact and easier to follow. This is not restricting, since $*:\mathtt{E} \equiv \mathtt{*seq(E}, \varepsilon)$, where $\varepsilon \equiv$ no event.

4.1 Event Satisfaction

We can now define the meaning of arbitrary EPL event expressions, through the notion of *satisfaction* of such expressions.

We start with qualified basic events. For instance, the satisfaction of the event `E = ins(ACC(X), X.Type = "Savings")` is defined as follows:

$$\begin{aligned}
&\texttt{ins_ACC}(\texttt{Accno},\texttt{Owner},\texttt{Type},\texttt{Bal},\texttt{Time},\texttt{J}) \leftarrow \\
&\qquad\qquad \texttt{hist_monit}(\texttt{ins},\text{“ACC”},\texttt{Time},\texttt{J}), \\
&\qquad\qquad \texttt{insertedACC}(\texttt{Accno},\texttt{Owner},\texttt{Type},\texttt{Bal},\texttt{Time}). \\
&\texttt{sat}_{\texttt{E}}(\texttt{Accno},\texttt{Owner},\texttt{Type},\texttt{Bal},\texttt{Time},\texttt{J}) \leftarrow \\
&\qquad \texttt{ins_ACC}(\texttt{Accno},\texttt{Owner},\texttt{Type},\texttt{Bal},\texttt{Time},\texttt{J}), \quad \texttt{Type} = \text{“Savings”}.
\end{aligned}$$

The predicate `ins_ACC` describes the history of occurrences of `insertedACC`; for each occurrence of this event type, `ins_ACC` contains a tuple with its attribute bindings and the stage of the occurrence.

In general, a qualified basic event is represented as $\texttt{E} = \texttt{evtkind}(\texttt{R}(\texttt{X}),\ \texttt{q}(\texttt{X}))$, where `q` denotes the event's condition expression, which can refer to the attribute values of tuple variable `X`. The rule template for the *satisfaction predicate* of such an event is:

$$\texttt{sat}_{\texttt{E}}(\texttt{X},\texttt{J}) \leftarrow \quad \texttt{evtkind_R}(\texttt{X},\texttt{J}),\ \texttt{q}(\texttt{X})$$

The concept of "an event *immediately following* another event" can also be expressed. Take for instance, the immediate sequence of Example 2, which is represented as:

$$\texttt{F} = (\ \texttt{upd}(\texttt{ACC}(\texttt{X}),\texttt{q}_1(\texttt{X})),\ \texttt{upd}(\texttt{ACC}(\texttt{Y}),\texttt{q}_2(\texttt{X},\texttt{Y}))\ )$$

Its semantics is defined by the following three $Datalog_{1S}$ rules (from now on, unless otherwise indicated, variables will denote tuples):

Example 5

$$\begin{aligned}
&\texttt{sat}_1(\texttt{X},\texttt{J}) \leftarrow && \texttt{upd_ACC}(\texttt{X},\texttt{J}),\ \texttt{q}_1(\texttt{X}). \\
&\texttt{sat}_2(\texttt{X},\texttt{Y},\texttt{s}(\texttt{J})) \leftarrow && \texttt{upd_ACC}(\texttt{Y},\texttt{s}(\texttt{J})),\ \texttt{sat}_1(\texttt{X},\texttt{J}),\ \texttt{q}_2(\texttt{X},\texttt{Y}). \\
&\texttt{sat}_{\texttt{F}}(\texttt{X},\texttt{Y},\texttt{J}) \leftarrow && \texttt{sat}_2(\texttt{X},\texttt{Y},\texttt{J}).
\end{aligned}$$

The first qualified basic event occurs at stage `J`, if an update on relation `ACC` is recorded at this stage and condition $\texttt{q}_1$ is satisfied. The second update on `ACC` must then occur at the next stage `s(J)` and condition $\texttt{q}_2$ must be satisfied (observe that $\texttt{q}_2$ can refer to the tuple variable `X` defined by the first basic event, in addition to `Y`). The third rule is a *copy-rule*, inasmuch as the satisfaction of composite event `F` coincides with that of $\texttt{sat}_2$.

There exists a natural mapping from EPL expressions to $Datalog_{1S}$. Thus, to formally define the meaning of an EPL expression, we only need to define a procedure which derives an equivalent set of $Datalog_{1S}$ rules for that expression. The resulting set of rules has a well-established formal semantics (model-theoretic and fixpoint-based) [3]. To formalize the translation, we represent EPL expressions by their parse trees, using the following prefix notation:

Since an EPL expression is evaluated with respect to a particular module, a separate *event history* must be obtained for each such module. Focusing now on one module, we assume that a relation evt_monit(EventType, TableName) is kept for it, that records the basic event types the module monitors. Then, our module's *event history* is defined by the following stratified *Datalog* rules:

```
hist_monit(nil, nil, 0000, 0.)
hist_monit(E, R, T2, s(J)) ← hist_monit(_, _, T1, J),
                             hist(E, R, T2), evt_monit(E, R),
                             ¬between(T1, T2).
between(T1, T2) ← hist(E, R, T), evt_monit(E, R), T1 < T, T < T2.
```

In this way, an *event history* can be defined for each module of interest. For instance, the following table contains a brief example event history for module AccMonitor:

hist_monit	EventType	TableName	TimeStamp	Stage
	nil	nil	0000	0
	upd	ACC	1423	1
	upd	ACC	1425	2
	ins	ACC	1430	3
	ins	ACC	1502	4

Observe that a sequence number, called *stage* has been introduced. The stage sequence defines an ordered structure on the distinct timestamps, that allows us to express properties of composite events that are based on the *relative order* of occurrence of their component basic events, as opposed to absolute time properties. Thus, the stage is the unit of time (*chronon*) in our model. Absolute time properties of events can also be expressed using their timestamps.

These observations lead naturally to the use of $Datalog_{1S}$ as a means for defining the semantics of EPL rules.

Different *event occurrence granularities* can be handled. At the "smallest database operation" granularity, every new insertion, deletion, or update creates a new stage. However, if "transaction boundaries granularity" is assumed, then each committed transaction creates a new stage, and all the basic events that occurred within this transaction are recorded in its stage, timestamped with the transaction's commit time. Basic events that share a sequence (stage) number are called *simultaneous* and are further discussed in Section 4.5.

As most active relational databases do, we further assume that for each DB table, there are three relations accumulating the inserted, deleted and updated tuples, together with their timestamps. For inserts into ACC for instance, we have the relation
ins_ACC(Accno, Owner, Type, Balance, Timestamp).
The del_ACC table has a similar format, while for updates, we must record both the old and new values:

upd_ACC(Accno_old, Accno_new, Owner_old, Owner_new, ... Timestamp)

We assumed here, that a transfer transaction results in an immediate sequence of updates. The condition expression of the second update can refer to the tuple variables of both basic events.

Example 3 *Report the cases where two large deposits are made to an account, without any intervening withdrawal from it.*

```
Good Customer:
   ( update(ACC(X), X.new_Balance-X.old_Balance > 100000),
     *:! update(ACC(Y),
                Y.Accno = X.Accno, Y.new_Balance < Y.old_Balance),
     update(ACC(Z),
            Z.Accno = X.Accno, Z.new_Balance-Z.old_Balance > 100000)
   -> write("Good Customer: %s \n", X.Owner)
```

Example 4 *Schedule a wake up call to happen at 7 am every morning, starting from next morning. H and D are symbolic constants, denoting Hour and Day, respectively.*

```
DemonInit:
   clock( nextday() + 7H )
   -> Wake Up Call ;
      SQL("update Demons set Dtime= %d where Name= 'WakeUp'", now() + 1D).

DemonUpdate:
   [ update(Demons(X)),  clock(X.new_Dtime) ]
   -> Wake Up Call ;
      SQL("update Demons set Dtime= %d where Name='WakeUp'",
            X.new_Dtime+1D).
```

The relation `Demons(Name, Dtime)`, contains the name and the time (initially set to 0) of the next occurrence of each *demon*. `DemonInit` schedules the first clock event to occur at 7 am the following day. (the *nextday()* built-in function returns the first second of the next day. At that time, a wake-up call action will be taken and an SQL update statement will be issued to set the *WakeUp* demon's *Dtime*, to the time instant of the next wake up call.

In the `DemonUpdate` rule, provided that the actual *update event* occurs within 24 hours, a new clock event will occur at the new time instant, and so on.

4 Semantics of EPL

We first introduce the notion of *event histories*, against which the EPL expressions are evaluated. The *global event history* is a series of *basic events*, that is ordered by time of occurrence (timestamp) and can be obtained from a system log. It can be represented by the relation `hist(EventType, RelName, TimeStamp)`, where each tuple records a basic event occurrence and contains its type (`insert`, `delete`, or `update`), the name of the relation upon which it occurred, and the time of the occurrence.

occurrence, to refer to the time instant when an event expression is *satisfied* (an instance of this event expression is completed).

The power of EPL follows from its ability to specify composite events. Composite event expressions are defined as follows:

Definition 1 *Let $E_1, E_2, ..., E_n$, $n > 1$, be EPL event expressions (maybe composite themselves). The following is also an EPL event expression :*

1. *$(E_1, E_2, ..., E_n)$: a sequence consisting of an instance of E_1,* immediately *followed by an instance of E_2, ...,* immediately *followed by an instance of E_n.*
2. *$* : E$: a sequence of zero or more consecutive instances of E.*
3. *$(E_1$ & E_2 & ... & $E_n)$: A conjunction of events. It occurs when all events $E_1, \ldots, E_n$ occur simultaneously.*
4. *$\{E_1, E_2, \ldots, E_n\}$: A disjunction of events. It occurs when at least one event among $E_1, \ldots, E_n$ occurs.*
5. *$!E$: It occurs when not any instance of E occurs.*

A number of additional (derived) constructs may be defined in terms of the basic ones (see also [12]). Some of these are :

- *$any \equiv$ The disjunction of all basic events (of the types monitored). It occurs every time such an event occurs.*
- *$[E_1, E_2, ..., E_n] \equiv (E_1, * : any, E_2, * : any, \ldots, * : any, E_n)$. Relaxed sequence. It consists of an instance of E_1, followed* later *by an instance of E_2, ..., followed* later *by an instance of E_n.*
- *$prior(E_1, E_2) = [E_1, any]$ & E_2. An occurrence of E_2 follows an occurrence of E_1 (i.e., an instance of E_1 is completed prior to the completion of an instance of E_2)*
- *$first(E) \equiv (E$ & $![E, any])$: It occurs, when the first instance of E occurs.*

Note that in an instance of $[\mathtt{E_1}, \mathtt{E_2}]$, the first basic event in the instance of $\mathtt{E_2}$ must follow an occurrence of $\mathtt{E_1}$, where in $\mathtt{prior}(\mathtt{E_1}, \mathtt{E_2})$, this is not required.

In addition to the above, a composite event may have attributes, which are derived from the attributes of its component basic events. Attribute semantics and scope rules are described in the next section. Examples of EPL composite event expressions follow.

Example 2 *Report transfers of large amounts, from a customer's savings account to his/her checking account.*

```
LargeTransfer:
   ( update(ACC(X),
            X.Type = "Savings", X.old_Balance-X.new_Balance > 100000),
     update(ACC(Y),
            Y.Type = "Checking", Y.Owner = X.Owner,
            Y.new_Balance-Y.old_Balance = X.old_Balance-X.new_Balance) )
   -> write("Large Transfer of %s \n", X.Owner)
```

The `AccMonitor` module keeps track of all the modifications in the `ACC` relation. Rule `LargeWithdrawal` specifies a *qualified basic event*. Such an event has the form:

$$\texttt{evtkind(}\ \texttt{Rname(X)},\ <\texttt{condition-expression}>\ \texttt{)},$$

where `X` denotes the tuple of relation `Rname`, that has been inserted, deleted or updated. For *update* events, EPL makes available to the programmer both the new and the old contents of the updated tuple. The prefixes `new` and `old` are used to distinguish between them. When no prefix is specified, the new contents are assumed.

The attribute `evtime`, which is attached to a basic event's tuple variable contains the time the event occurred. The < condition-expression > is built using the standard arithmetic and comparison operators and the logical connectives AND, OR and NOT.

Actions: There are three kinds of actions : (a) Write actions, (b) SQL commands and (c) Operating system calls.

In all cases, the format of the action specifier is similar to that of the *printf* statement in the C language. The action's arguments are taken from tuple variables defined by basic events in the rule's body.

Negated Events: A (qualified) basic event may be negated. Consider for instance the following rule, as part of the module `AccMonitor`:

```
NoUpdateOn00201:
      ! update( ACC(X), X.Accno = 00201 )
      -> write("any event but an update on 00201")
```

This *negated qualified basic event* will be satisfied by the occurrence of *any* basic event of the types monitored by the `AccMonitor` (including possible insertions and deletions), except for an update of account 00201.

Clock Events: EPL allows *explicit time events*, which are called *clock events*. Such an event has the form `clock(TE)`, where `TE` is a *temporal expression* that specifies the time when the clock event occurs. Absolute or relative time constants specified up to *second granularity* can be used in temporal expressions.

Clock events are very useful in several applications. They can be used for instance to enforce deadlines, or to schedule periodical temporal events. Such an example is shown in the next section.

Simple Events: A simple event is a *basic event* (which may be qualified), or an *explicit time (clock) event*. A basic event may be negated, but a clock event cannot.

3.2 EPL Language Constructs

We distinguish between *event expressions* (also called *event types*) and *event instances*. An event expression `E` is specified using the EPL language, where an event instance of `E` consists of a sequence of basic events that participated in the satisfaction of `E`. In the sequel, we will refer to *events* when the distinction is clear from the context. We will also use the term *event*

Therefore, $Datalog_{1S}$ provides a general method for defining composite event languages and a tool for understanding and comparing the constructs and expressive power of active database languages dealing with composite events.

The method here proposed has some similarities to Chomicki's work on enforcing complex integrity constraints [6]. The constraints which are expressed in Temporal Logic, are enforced by derived *condition-action* rules. A more recent approach is that of [22], which employs another variation of Temporal Logic, where assignment operators are used instead of first order quantifiers. In [22], temporal aggregates are also considered. The *condition-action* rules used in these approaches are contrasted to our *event-driven* rules.

3 The Event Pattern Language

In the rule-based syntax of EPL, sequences of goals specify sequences of events. Events can be *simple* or *composite*, constructed using sequences, repetitions, conjunctions, disjunctions and negations of other events. EPL is implemented as a portable front-end to active relational databases supporting simple event detection. Two versions of EPL have been developed, the first in $\mathcal{LDL}$++ [2] and the second in CLIPS [18].

Because of space limitations, we concentrate here on the composite event specification of EPL and we omit any discussion of rule processing and management.

3.1 EPL Programs

An EPL program is organized in *modules*, which can be compiled and enabled independently. The *events declaration section* of a module defines the set of relevant basic event types, monitored by the module. A *basic event type* is denoted as :

```
insert(Rname), delete(Rname), update(Rname),
```

where `Rname` is the name of a database relation.

EPL rules are specified in a module's *rules section*. Each rule has a name, which is unique within its module. A rule's body (head) corresponds to an event (action). Example 1 demonstrates an EPL module with one rule. In all examples, we use the following bank accounts relation.

```
ACC(Accno, Owner, Type, Balance)
```

Example 1 *An EPL module with a rule that keeps track of large withdrawals from savings accounts.*

```
begin AccMonitor
   monitor  insert(ACC), update(ACC), delete(ACC);
   LargeWithdrawal:
      update( ACC(X),
              X.Type = "Savings", X.old_Balance-X.new_Balance > 100000 )
   -> write( "Large withdrawal from account %d at time %s \n",
              X.Accno, asctime( X.evtime) ).
end.
```

1. FSMs do not support variables, and thus, they cannot detect parameterized events, which are essential in most real-life applications. It is suggested in [12], that parameterized events are handled, by augmenting the states of an FSM with sets of partial variable bindings of the composite event that the FSM implements (detects). The resulting model surrenders the initial simplicity and intuitive appeal of FSMs, without providing a fail-proof formalism. In particular, the semantics of negation when attributes are involved remains a problem.

2. Since FSMs are inherently sequential, simultaneous events cannot be supported, unless transitions based on combinations of events are allowed. But even if simultaneous events are of no concern, constructs such as conjunction can only be modeled by an exponentially increasing size of states in an FSM. This is because the FSM for $\mathtt{E} = \mathtt{E}_1 \wedge \mathtt{E}_2$ is built by constructing the cross-product of the FSMs for $\mathtt{E}_1$ and $\mathtt{E}_2$ [14].

In SAMOS, Petri nets are used instead for the detection of composite events [11]. Petri Nets solve the problem of exponential blow-up in FSMs, by allowing concurrent processing — actually, this was one of the main reason for the development of the Petri Nets theory. Also, *coloured Petri Nets* [7] cater for the handling of parameterized composite events. However, there are still some limitations. Simultaneous events cannot be handled. Also, it is not clear how the semantics of general negation (as defined in the next section) can be captured by Petri nets. The composite event detection mechanism of Snoop, which is based on Event Graphs [4] suffers from similar limitations.

A simpler approach to overcome the afore-mentioned problems is that of using $Datalog_{1S}$ [3]. This is a temporal language that extends *Datalog*, by allowing every predicate to have at most one temporal parameter (constructed using the unary successor function s), in addition to the usual data parameters. The temporal parameter in our case models the succession of states in the event history, and it is called the *stage* argument.

For instance, all the occurrences of the pattern `E = (f,g)` can be detected by the following two $Datalog_{1S}$ rules:

$$\begin{aligned} \mathtt{in}_1(\mathtt{J}) \leftarrow \quad & \mathtt{evt_hist}(\mathtt{f}, \mathtt{J}) \\ \mathtt{sat}_\mathtt{E}(\mathtt{s}(\mathtt{J})) \leftarrow \quad & \mathtt{evt_hist}(\mathtt{g}, \mathtt{s}(\mathtt{J})),\ \mathtt{in}_1(\mathtt{J}). \end{aligned}$$

The rules are applied against an *event history* relation `evt_hist`, where each event is sequentially numbered, as shown in Figure 1. Observe here that $sat_E(s(J))$ is true at each occurrence of the pattern `(f,g)`. Furthermore, it is trivial to support parameters, by simply adding parameter columns to the *event history* relation, and adding corresponding variables to the $Datalog_{1S}$ predicates. One such parameter can be e.g., an explicit time-stamp denoting when the event occurred. Thus, explicit time-based conditions can be supported and so is the notion of temporal events. In a nutshell, $Datalog_{1S}$ provides a simple and powerful formalism, where all the ingredients needed for the definition of composite event patterns are already in place. One key ingredient is in fact the extraordinary semantics foundations of Datalog, where along with an abstract least-model semantics, we have the equivalent fixpoint-based semantics and various operational improvements, such as differential fixpoint and magic sets.

formalisms. Consequently, it is desirable that a general more abstract formalism is employed, where correctness testing, as well as optimization and extensibility studies can be performed.

By contrast, powerful semantic foundations are available for the rules of deductive databases [1, 15]; so far though, these have had little impact on the design of active databases. In this paper, we show that this omission is quite unfortunate, since $Datalog_{1S}$ provides a natural and powerful basis for the definition of the semantics of composite-event-triggered active database rules. The general method we propose is based on a syntax-directed translation of such active rules into $Datalog_{1S}$, whose formal semantics is then used to define the meaning of the original rules.

We first demonstrate the method by giving a complete definition of Event Pattern Language (EPL). EPL is the language of an active database system designed and implemented at UCLA, which provides the capability of detecting, reasoning upon and acting on complex patterns of events that evolve over time. Then, we explain how our method can be applied to ODE [13], Snoop [5] and SAMOS [10], in order to formally define the intuitive and/or operational semantics of the composite event specification languages of these systems.

The method proposed in this paper matches and surpasses the capabilities of the formalisms used in the past, in terms of defining precisely the semantics of different language characteristics, including *event attributes, negation, simultaneous events, explicit time events and parameter contexts*. Therefore, we obtain a general method for defining the semantics of such languages, and a useful tool for comparing their constructs and their expressive power.

2 Motivations

Among the various formalisms that have been used in the past for implementing composite event detectors, the Finite State Machines (FSMs) used in ODE is probably the simplest and most familiar one.

Figure 1 shows a sample *event history* and an FSM that can recognize the *first* occurrence of a composite event, but not any latter occurrences of it. `f` and `g` denote two basic events and `(f,g)` denotes an immediate sequence, i.e., event `g` must immediately follow event `f`.

In ODE, for each composite event type `E`, a general non-deterministic automaton is created, which can succeed multiple times, once for each occurrence of `E`. Even so, FSMs suffer from two major limitations:

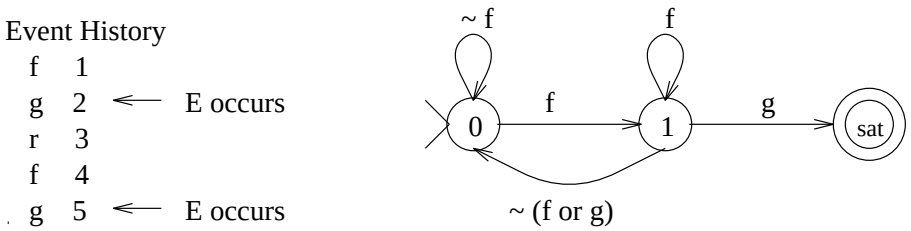

Figure 1: An FSM for the composite event E = (f, g). 0 and *sat* denote the *start* and *success* states respectively.

A Formal Semantics for Composite Temporal Events in Active Database Rules *

Iakovos Motakis Carlo Zaniolo
Computer Science Department, University of California
Los Angeles, California 90024
motakis@cs.ucla.edu zaniolo@cs.ucla.edu

Abstract

A major thrust of current research in active databases is allowing complex patterns of temporal events to serve as preconditions for rule triggering. Currently, there is no common formalism for specifying the semantics of composite events specification languages and different systems use an assortment of approaches including Finite State Machines, Petri Nets and Petri Nets. In this paper, we propose a unifying approach, which is based on a syntax-directed translation of the composite event expressions into $Datalog_{1S}$, whose formal semantics is then used to define the meaning of the original event expressions. We demonstrate our method by providing a formal specification of the Event Pattern Language (EPL) developed at UCLA. This method overcomes problems and limitations affecting previous approaches and is applicable to other languages such as ODE, Snoop and SAMOS—thus, allowing a more direct comparison across different languages.

1 Introduction

A new generation of database systems supports active rules for the detection of events occurring in the database and the triggering of induced actions. In many applications, the simple-event detection mechanisms found in commercial systems, such as Ingres, Oracle or Sybase, are not sufficient; complex sequences of events must instead be detected as the natural precondition for taking actions and firing rules [9]. Sophisticated research prototypes have in fact been implemented recently to provide this capability— an incomplete list of such systems includes [9, 13, 10, 5].

A difficult problem facing the designers is that of specifying the semantics of active rule languages. Indeed, giving a formal semantics to active database behavior presents a challenge even when only simple events are involved [20, 19]. For composite event expressions, this problem becomes even more complex, inasmuch as the issues of temporal databases and active rule languages are now combined and intertwined [21].

In previous systems, the operational semantics of composite event detection has been defined using widely different frameworks, such as Finite State Automata [12], Petri Nets [11], or Event Graphs [4]. Eventhough these systems have captured a great deal of the functionality required by active database applications, they are restricted by the limitations of their underlying

*This work was partially supported by NASA HPCC grant NAG 5-2225.