Object Histories as a Foundation for an Active OODB

Johan van den Akker and Arno Siebes
CWI, P.O.Box 94079, 1090 GB Amsterdam, The Netherlands
e-mail: {vdakker,arno}@cwi.nl

To appear in the Proceedings of the DEXA'96 workshop.

Abstract

Several links exist between active and temporal databases. These are summarised by the observation that rules are triggered by a specified evolution of the database. In this paper, we discuss the relation between active and temporal database using DEGAS, an object-based active database programming language. To achieve full active database functionality, a DEGAS object records its complete history. Hence, all data needed for a temporal database supporting a single temporal dimension is provided. Furthermore, the semantics of the active behaviour of DEGAS are defined straightforwardly in terms of the object history. Finally, we discuss the advantages and disadvantages of extending DEGAS with a second time dimension (to achieve full temporal functionality) from an active database perspective.

1 Introduction

In recent years, the areas of active databases [19] and temporal databases [17] have been the focus of a significant research effort. Temporal databases concentrate on recording and querying database states relative to time. Active databases add dynamic behaviour to data in the form of rules. A rule defines an action which is executed when specified events occur in the database. In other words, the action is executed, if the database state evolves over time in the specified way. Hence, an investigation of the incorporation of temporal elements in an active database is justified.

From an active database viewpoint, two questions with regard to its temporal functionality are of interest. The first is, which record of temporal data is required by a full active database. The second question is, what are the benefits of incorporating full temporal database functionality in an active database.

To answer these questions, this paper presents the temporal element of DEGAS [2], an active object-oriented database programming language. The incorporation of a temporal element is achieved by including the complete history of an object in its state. As a consequence, the semantics of the active behaviour of a DEGAS object can be specified in terms of the object history, using a standard process algebraic specification formalism.

In this paper, we first identify the temporal elements of an active database. Then, we give a short introduction to the main concepts of the DEGAS data model. After that, the history of a DEGAS object is defined. Following this, we show the formalisation of the active behaviour of an object in terms of the object history. Finally, we compare our approach to other work in temporal and active databases.

2 Time in Active Databases

The key feature of active databases [19] are production rules. Usually, these are defined as event-condition-action (ECA) triples. The event specification may be a complex event expression composed of multiple basic events, such as method calls [7, 10] and time events [7, 13, 10]. Since rule definitions specify sequences of events over time, an active database has an inherent temporal element, as observed by Dittrich [8] and Widom and Ceri [19].

We can also see this by looking into rule triggering in more detail. In order to detect complex events, we need to store the basic events occurring in the database. These make up an event queue or event pool. Since a complex event expression usually specifies a sequence of events, the record of basic events must store information about the order in which events occurred.

This inherent temporal element in active databases gives rise to the question of the relation to databases that keep historical data. To that end, we examine what temporal data needs to be stored in an active databases. Not surprisingly, this depends on the rule language.

Many active databases include time in an event expression. This can be in relative form, such as "5 days after event *A*" or absolute such as "every day at midnight". Orthogonally, we can put time in event specifications in different ways. We can add a time parameter to all events or we can have explicit time events in the event specifications. The latter choice will make a difference in the way we check the temporal part of the rule specification. In the former

case, we can check temporal conditions in the condition of the rule. In the latter, the time events are included in the event detection mechanism.

Since most active database management systems offer the possibility to specify parameters of events, we also need to store the parameters of a method call in addition to the time it occurred. In this way a rule can be triggered on method calls only for certain parameter values. For example, we may have a rule on a bank account that is only invoked if a debit action of more than 1000 guilders is executed.

Every extension of event specification in the definition of rules beyond single basic events necessitates a record of part of the history of the database. If we wish to offer all facilities described above in an active DBMS, i.e., time in event specifications and parameters to events, we have to store all method calls with their parameters and timestamps. It is obvious that we can reconstruct all historical states of the database, if we have all state transitions in the form of method calls. Hence, it is a small step from an active database to a historical database.

A historical database is a restricted form of temporal database. Temporal databases [17] record data relative to time. A full temporal database has two temporal dimensions. Valid time denotes the time a value held in the real world. Transaction time denotes the time a value was entered into the database. The combination of these two dimensions allows us to alter data retrospectively, for example, to correct errors. Historical database are temporal databases with only one temporal dimension. In other words, a historical database only records the states a database went through over time.

The DEGAS model presented in this paper aims to incorporate temporal functionality in an active database. To this end, the state of a DEGAS object includes its history, i.e., a record of past states and method calls. Consequently, the semantics of active rules in a DEGAS object are defined in terms of its history.

3 The DEGAS data model

We now give a concise introduction to the main concepts of the DEGAS data model. It is based on autonomous objects. The motivation for object autonomy is on one hand a natural further development of active object-oriented databases and on the other hand the development of highly distributed information systems. The main contributions of DEGAS are:

- The integration of historical and active database functionality.
- A straightforward mechanism for object evolution, especially suited for implementing roles.
- Complete encapsulation of an object's behaviour, including rules.
- A good formalisation of rule semantics.
- A conceptual model for distributed information systems.

For a more elaborate introduction of DEGAS the reader is referred to [2]. A full formal definition of DEGAS can be found in [1].

The fundamental notion in DEGAS is the object. The definition of an object in DEGAS consists of structure and behaviour. The structure of an object is defined by its attributes. The behaviour definition of a DEGAS object consists of three elements: methods, lifecycles and rules. Methods define the actions an object can execute. The lifecycle of an object specifies sequencing and preconditions of methods. A rule states that an object will execute a given action in certain situations, specified by events and conditions on object states.

In other words, methods and lifecycles define the *potential* behaviour of an object, whereas rules describe its *actual* behaviour as far as can be pre-determined. Conventionally, only potential behaviour is specified in an object.

Figure 1 shows an example DEGAS object modelling a PIN card. Attribute and method specification is straightforward in DEGAS. Lifecycles are guarded basic process algebraic expressions [3] composed from the set of method names as basic actions using the sequential composition (;), alternative composition ($+$), repetition (*), and parallel merge ($||$) operators. For example, the third line of the lifecycle definition in our example specifies that a *ReqWithdraw* action must be followed by a *WithdrawOK* or a *WithdrawRefuse* action, and that this sequence may be repeated arbitrarily. The parallel merge operator $||$ means that two actions take place without restriction on their sequence, i.e., $A||B = A;B + B;A$.

Rules in DEGAS follow the usual Event-Condition-Action (ECA) format. The informal semantics of an ECA rule is, that if the event occurs and the object satisfies the condition, the action is performed. In DEGAS events are specified the same as lifecycles with addition of a negation operator ($\neg$). Conditions in lifecycles and rules can refer to historical values of attributes. If an attribute name is parameterised by a timestamp, it refers to the value of the attribute at the specified time. Otherwise, it refers to the current value of the attribute. The rules in PINcard show historical references in DEGAS rules.

More in particular, the first rule specifies that the PINcard sends its permission for a cash withdrawal after a request, if the total amount withdrawn during the preceding week is less than the limit of the card. The second rule responds with a refusal, if the limit is exceeded.

The class of a DEGAS object specifies its inherent *capabilities* (= attributes, methods, lifecycles and constraints). Object specialisation in DEGAS is achieved through addons. An addon models transient capabilities of an object. Addons can be added to and deleted from an object dynamically, for example, when an object engages in a relation. A restricted form of inheritance is supported by DEGAS. Since this is not relevant for this paper, the interested reader is referred to [1] for more details. Relations in DEGAS are also objects with structure and behaviour.

```
Object PINcard
Attributes
   number : integer
   limit : integer
   account : Oid
   issuer : Oid
   owner : Oid
   PIN : integer
Methods
   ReqWithdraw(amount:integer,requester:Oid) = {

   }
   WithdrawOK(amount:integer,requester:Oid) = {
      requester.allowed(amount)
   }
   WitdrawRefuse(amount:integer,requester:Oid) = {
      requester.refuse(amount)
   }
   ChangeLimit(newLimit : integer) = {
      limit = newLimit
   }
   ChangePIN(newPIN : integer) = {
      PIN = newPIN
   }
Lifecycles
   ([sender==issuer] ChangeLimit)*
   ([sender==owner] ChangePIN)*
   (ReqWithdraw;(WithdrawOK + WithdrawRefuse))*
Rules
   On (WithdrawOK(amount,atm)(t))*;
      ReqWithdraw(reqAmount,machine)(t1)
      if t1 − Min(t) ≤ 1 week
         && Sum(amount, t)+reqAmount ≤ limit
      do WithdrawOK(reqAmount,machine)
   On (WithdrawOK(amount,atm)(t))*;
      ReqWithdraw(reqAmount,machine)(t1)
      if t1 − Min(t) ≤ 1 week
         && Sum(amount, t)+reqAmount > limit
      do WithdrawRefuse(reqAmount,machine)
EndObject
```

Figure 1: A DEGAS object

In the rest of the paper, we focus on the formalisation of the active behaviour of a DEGAS object in relation to its historical record. Hence, we do not discuss the aspect of object evolution through addons.

4 The History of an Object

In the previous section we informally introduced the main concepts and the syntax of DEGAS and presented its temporal functionality. We now proceed with the formalisation of the relevant part of the DEGAS data model. In this section we give a formal definition of the state of a DEGAS object, which consists of its complete history.

Object typing in DEGAS follows Cardelli [5] and Balsters [4]. The underlying type of an object is a tuple type containing the attributes. Besides simple types, such as Integer, String or Oid, there are set and tuple types. The underlying type of an object definition contains at least its own identifier *this*.

An operator *Type(D)* can be applied to an object definition to obtain the underlying type of the object. For example, the underlying type of the PINcard object from Section 3 is

$$\begin{aligned}&\mathit{Type}(\text{PINcard}) = \\ &\langle \text{this:Oid, number:integer, limit:integer, account:Oid,} \\ &\quad \text{issuer:Oid, owner:Oid, PIN:integer}\rangle\end{aligned}$$

Following temporal database terminology, the state of an object at a certain point in time is called a *snapshot state* [14]. It records the time the object came in this state, a valuation for the attributes and the method call that brought the object into this snapshot state.

More formally, a snapshot state of an object O is a triple $\langle t, I(\tau), MC\rangle$, where t is a timestamp giving the start time of the validity of this state, $I(\tau)$ is the valuation of $\tau = \mathit{Type}(O)$ of the attributes in this state and MC is a method call, which consists of a method name and a parameter list.

State History The state history of a DEGAS object records the snapshot states the object went through during its existence. This means that a state history SH is a sequence of snapshot states:

$$SH = SH(0); SH(1); \ldots; SH(n)$$

where $\forall i, 0 \leq i \leq n-1 : t_i < t_{i+1}$. This definition of an object history is largely similar to that found in [12]. The main difference is that a DEGAS object history deals directly with DEGAS methods calls and attributes, instead of the more abstract notions of actions and input and evaluation attributes. The following part of the history of a PINcard

object is an example:

$$
\begin{array}{l}
\vdots \\
\langle 12:34:00, \\
\quad \langle \text{This} = 102040, \text{limit} = 500, \ldots, \text{PIN} = 1234 \rangle, \\
\quad \text{ChangePIN}(1234) \rangle; \\
\langle 13:45:00, \\
\quad \langle \text{This} = 102040, \text{limit} = 1000, \ldots, \text{PIN} = 1234 \rangle, \\
\quad \text{ChangeLimit}(1000) \rangle \\
\vdots
\end{array}
$$

Lifecycles and the event expressions of rules are checked using a projection of the state history, the event history. It only contains timestamp-method call pairs. If we have a state history $SH = SH(0); SH(1); \ldots; SH(n)$, then the event history EH is the sequence $EH(0); EH(1); \ldots; E(n)$ of time-event pairs, where:

$$
\begin{array}{l}
\forall i, SH(i) = \langle t_i, \tau_i, I(\tau_i), MC_i \rangle : \\
\qquad EH(i) \stackrel{def}{=} \langle s_i, e_i(p_1, \ldots, p_m) \rangle, \\
\qquad s_i = t_i \wedge e_i(p_1, \ldots, p_m) = MC_i
\end{array}
$$

The example state history above gives us this event history:

$$
\begin{array}{l}
\ldots \\
\langle 12:34:00, \text{ChangePIN}(1234) \rangle; \\
\langle 13:45:00, \text{ChangeLimit}(1000) \rangle \\
\ldots
\end{array}
$$

5 Active Behaviour

The state history of a DEGAS object serves as a basis for the formalisation of its active behaviour. Hence, we can formulate the semantics of lifecycles and rules in terms of process algebraic expressions relative to the observed execution of the object.

Lifecycle Composition Execution of methods and rules must conform to the lifecycles on the object. In addition rules are triggered by the contents of the event history. As we saw above, lifecycles are guarded basic process algebraic expressions [3] with the set of methods of the object as its basic actions. Hence, the semantics of lifecycles is also formulated in process algebraic terms. Suppose we have an object O with the following lifecycle definition:

Lifecycles
C_1
C_2
$\vdots$
C_n

Then O follows the process:

$$C = C_1 | C_2 | \ldots | C_n$$

with communication function γ defined by: $\forall \alpha \in \mathcal{M} : \gamma(\alpha, \alpha) = \alpha$, where $\mathcal{M}$ is the set of methods of O.

In process algebra in general, a communication function γ specifies synchronisation between two processes. $\gamma(A, B) = C$ means that the actions A and B have to take place simultaneously and are replaced in the trace of the process by the single action C. For example, if we have the process $(A; B)|(C; D)$ and $\gamma(B, D) = E$, then a resulting trace might be: $A; C; E$.

In practical terms the communication function defined for a DEGAS object means that, if an action occurs in more than one lifecycle, the execution of that action is a step forward in all lifecycles. For example, the lifecycle of the PINcard object from Section 3 is:

$$
\begin{array}{l}
([\text{sender} == \text{issuer}]\text{ChangeLimit})^* \\
\quad \| ([\text{sender} == \text{owner}]\text{ChangePIN})^* \\
\quad \| (\text{ReqWithdraw}; (\text{WithdrawOK} + \text{WithdrawRefuse}))^*
\end{array}
$$

Since the three lifecycles defined in PINcard do not have actions in common, the communication merge $|$ reduces to an ordinary parallel merge $\|$. Note that a finite automaton is sufficient to match the event history with an event expression or a lifecycle.

Lifecycle Checking A method is executed if it does not violate the lifecycles imposed on the object. This means that the object state must satisfy the, possibly empty, precondition given by the lifecycle. In addition the method call in combination with the event history must match the event expression given. If this is the case, the method call is executed and appended to the event history. If the method call does not satisfy the lifecycle of an object, it is discarded.

The formalisation is, that a method call $MC = m(q_1, \ldots, q_k)$ is executed on an object O with state history $SH = SH(0); \ldots; SH(n)$ at time t, iff the lifecycle checking automaton induced by the lifecycle of O is in a state with an outgoing transition labelled with a method name μ and a condition C, such that, $m = \mu$ and C is true in O at time t. The resulting new state of the object is

$$SH' = SH; \langle t, M(m(q_1, \ldots, q_k), I(\tau)), MC \rangle$$

Here $M(m(q_1, \ldots, q_k), I(\tau))$ denotes the result of the execution of $m(q_1, \ldots, q_k)$ on attribute valuation $I(\tau)$.

Rule Triggering As explained above, a rule in DEGAS is an event-condition-action triple $\langle E, C, A \rangle$, where E is a basic process algebraic expression, C a condition and A a method call. An $\langle E, C, A \rangle$ is specified as follows in DEGAS:

On E
if C
do A

Before we define the execution of rules, we first give an informal sketch of the execution model of a DEGAS object. Basically, the object executes a cycle of two activities. There is a queue of method calls waiting to be executed. The object

takes a method call from this queue, checks if it is allowed by the lifecycles, and if so executes it. If the method call does not agree with the object's lifecycle, it is not executed and discarded. During the execution of a method a number of rules may be triggered. These are collected in a set of triggered rules. After a method has finished, a rule is picked at random from this set for execution. After the execution of this rule, we again construct a set of triggered rules to pick a rule from. If no more rules are triggered, we start again with method execution.

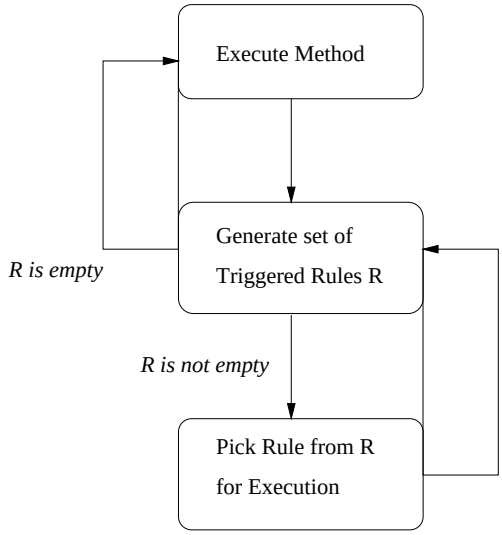

Figure 2: Execution Cycle of a DEGAS Object

From this informal description, we learn that the execution of rules is a two phase process. During a method call a set of rules that are triggered by that method call is built up. After the method has finished, rules are picked at random from this set. If the condition of the picked rule holds, the rule is executed.

A rule is triggered by an event occurring in the event history as a result of a method call. If we have an event history $EH = EH(0); \ldots; EH(m)$ and a method call $MC = \mu(p_1, \ldots, p_k)$ at a time t. Rule $R = \langle E, C, A \rangle$ is triggered at time t, if E parses the new event history $EH; \langle t, \mu(p_1, \ldots, p_k), t \rangle$ correctly.

After each execution of a method μ the set of triggered rules $\mathcal{R}_\mu$ is constructed. This means that the execution of a complex method, i.e., a method composed of method calls, is interrupted by the execution of rules triggered by the methods it calls.

After a method has finished, one rule is picked for execution from the set of triggered rules at random. If its condition is true and the method call that is its action satisfies the lifecycle of the object, it is executed. Otherwise, another rule is picked for execution. During rule execution rule triggering continues. In other words, after each method executed as a consequence of a rule, a new set of triggered rules is constructed.

Rule Execution Given a rule set $\mathcal{R}_\mu$ after the execution of method μ on an object O. The rule execution phase follows the algorithm:

1. $\mathcal{R} = \mathcal{R}_\mu$
2. A rule $R = \langle E, C, A \rangle$ is picked at random from $\mathcal{R}$ at time t.
3. If C is true and A satisfies the lifecycles of the object, A is executed. Otherwise, discard R and goto step 2.
4. Generate a new set $\mathcal{R}$ of rules triggered by the action of R.
5. If $\mathcal{R} \neq \emptyset$, then goto step 2.

This execution model gives us flexibility in the way rules interact with the lifecycles. If the action of a rule cannot be executed, because it violates the lifecycles of the object, we have two options. The action can be discarded or it can be retried at a later time.

The former will be used in situations where only a timely reaction is useful. These can be found in applications in financial markets. For example, our reaction to a falling price of shares is that we buy some. This is only profitable if we do it immediately, because otherwise the price may already have risen again. The other strategy is of use if the action of a rule must always be executed once a rule has been triggered. Rules that maintain integrity constraints will use such a strategy.

The standard behaviour of DEGAS is the immediate reaction. Suppose an object must react to the occurrence of the event $A; B$ with an action μ. This is specified by the following rule. If this rule is not chosen for execution after $A; B$, since another rule was chosen from the set of triggered rules, it is not considered again until the next occurrence of $A; B$.

On $A; B$
do μ

The alternative strategy can be programmed in a DEGAS rule through the non-occurrence operator $\neg$. The following rule, which specifies that the action is always triggered after the occurrence of $A; B$ until μ has occurred:

On $A; B; \neg\mu$
do μ

6 Comparison to Other Work

Temporal Databases If we compare the temporal functionality of DEGAS with temporal databases in general, the first thing to note is that DEGAS only records transaction time. Hence, it does not offer the functionality of databases which additionally include valid time. Therefore, we choose TSQL [15] for a comparison of DEGAS, since this is a straightforward mono-temporal extension of SQL. The temporal functionality offered by TSQL are WHEN-clauses, retrieval of timestamps, temporal ordering, a TIME-SLICE operation, and aggregate-functions.

WHEN-clauses specify temporal conditions on the data involved in a query, for example, an overlap in time of two values. Retrieval of timestamps means that we can ask the database for the time a condition was valid. Temporal ordering gives us the possibility to ask queries like "The first time employee John's salary was above 50K". All data needed to answer these three categories of queries is present in the history of a DEGAS object, since each attribute valuation is related to a time interval.

A time-slice restricts a query to a sub-interval of the complete history. Clearly, we can do this in DEGAS by evaluating a query on a subsequence of the object history only. Aggregates are functions applied to a time interval. Examples are average, minimum and maximum values of an attribute over a period of time. Aggregates relative to time can be considered a special case of time-slicing, since we calculate the aggregate over a specified time slice.. Hence, the data needed is present in a DEGAS object.

The comparison between TSQL and DEGAS is summarised by the following table. We see that a DEGAS object contains all the data needed for a full mono-temporal query language.

TSQL Feature	Information present
Conditional: WHEN	Y
Retrieval of timestamps	Y
Temporally ordered information	Y
Time-slices	Y
Aggregates	Y

DEGAS can be extended with a second temporal dimension, valid time, by qualifying every reference to time in a rule with the temporal dimension it refers to. For example, if we refer to a historical attribute value in the condition of a rule, we must specify whether it refers to valid time or transaction time.

The situation, however, gets more complex, if we need to define a rule execution model in an environment where temporal data are altered retrospectively. Suppose a rule is triggered on two events that happen within five minutes. If we specify that these five minutes are valid time, we must also consider changes of our knowledge of the past. If we find out later that two events happened within five minutes of each other, we must specify if it still makes sense to trigger the rule. Further complexity is introduced by the combination of valid timestamps and transaction timestamps.

As we saw earlier, a full active database requires the presence of all historical data needed to provide transaction time functionality. The incorporation of a second temporal dimension means a big complication of the rule model. At present a good solution to handle this complexity is still an open problem. Hence, we restrict DEGAS to a single temporal dimension.

Active Temporal Databases Previous research on the integration of temporal and active databases is the work by Etzion, Gal, and Segev [9]. Their model is rich on the temporal side, including three temporal dimensions. Active rules, however, are restricted to derived data. The main focus is on the consequences of changes in the rules specifying the derived data. An example given in [9] is a retroactive change in the way fines for speeding are calculated. The possibility of such changes necessitates the third time dimension in addition to valid and transaction time, decision time. This timestamp records the time derived data was calculated.

Temporal Specification of Triggers Another approach concerned with time in active databases is the work on temporal triggers by Sistla and Wolfson [16]. These are condition-action rules, where the condition is formulated in two variants of a temporal logic, called the Future Temporal Logic (FTL) and the Past Temporal Logic (PTL). These logics specify relations between timestamped database states. The basic modal operators are `Until` and `Nexttime`, thus allowing for the specification of durations. Temporal triggers address the temporal part of condition specification, but do not involve event specifications. We can only specify an action to occur when two events happen within a certain time interval by stating in the condition the effects of those events on variables in the database.

Active Databases One of the questions we tried to answer in this paper, is how the an active database benefits from the integration of temporal database functionality. The main benefit is found in the original motivation, viz., the possibility of using temporal conditions and event specifications in rules. Although this benefit is widely recognised, DEGAS is the first active database model to include a full record of historical data. Furthermore, the history allows the use of process algebra both for the definition of rule and lifecycle semantics, and for event and lifecycle specification. This can be contrasted with operational semantics found, for example, in Chimera [6] and Ariel [13]. It is also simpler than the denotational semantics of the Starburst

rule system [18]. Event histories are also used to define the semantics of event expressions in ODE [11], but here the relation with temporal databases is not considered.

Another advantage of the use of process algebra is that the event expression need not be converted to another formalism for the definition of the semantics. An example is the conversion of event expressions to Petri nets in SAMOS [10].

7 Concluding Remarks

In this paper we discussed the integration of temporal and active databases in the DEGAS model. We saw that a full active database requires the presence of the history of the database. In DEGAS historical data are available through the inclusion of the complete history in an object. This historical record contains all information necessary to implement a full transaction-time temporal query language. Furthermore, the object history facilitates a clear process algebraic definition of the active database semantics.

The incorporation of full temporal database functionality in an active database greatly increases the complexity of the rule execution model. A method to tackle this complexity still has to be found. Hence, DEGAS does not incorporate valid time.

A prototype implementation of DEGAS is under construction. The simplicity of the DEGAS rule model is expected to facilitate a performant system. Since the motivation of DEGAS is largely found in distributed information systems, further research will focus on a query model tailored to a database of distributed objects. Clearly, this query language will include facilities to exploit the historical data stored in DEGAS objects.

References

[1] J.F.P. van den Akker and A.P.J.M. Siebes. DEGAS: A temporal active data model based on object autonomy. Technical Report CS-R9608, CWI, Amsterdam, The Netherlands, 1996. Available through WWW (`http://www.cwi.nl/~vdakker/`).

[2] Johan van den Akker and Arno Siebes. DEGAS: Capturing dynamics in objects. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Advanced Informations Systems Engineering - Proc. of CAiSE'96*, pages 82-98, Heraklion, Crete, Greece, May 1996. Springer. LNCS 1080.

[3] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, 1990.

[4] Herman Balsters and Maarten M. Fokkinga. Subtyping can have a simple semantics. *Theoretical Computer Science*, 87:81-96, 1991.

[5] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Proceedings of the International Symposium on the Semantics of Data Types*, pages 51-68, Berlin, Germany, 1984. Springer.

[6] Stefano Ceri et al. *Active Rule Management in Chimera*, chapter 6 in [19].

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

[8] Klaus R. Dittrich and Stella Gatziu. Time issues in active database systems. In *Proc. of the Intl. Workshop on an Infrastructure for Temporal Databases*, Arlington, TX, USA, 1993.

[9] Opher Etzion, Avigdor Gal, and Arie Segev. Retroactive and proactive database processing. In J. Widom and S. Chakravarthy, editors, *Proc. of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems (RIDE-ADS)*, pages 126-131, Houston, TX, USA, 1994. IEEE Computer Society Press.

[10] Stella Gatziu, Andreas Geppert, and Klaus R. Dittrich. Integrating active concepts into an object-oriented database system. In Paris Kanellakis and Joachim W. Schmidt, editors, *The Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data*, pages 399-415. Morgan Kaufmann, 1991.

[11] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In M. Stonebraker, editor, *Proc. of the 1992 ACM SIGMOD Intl. Conf. on the Management of Data*, pages 81-90, San Diego, USA, 1992.

[12] Seymour Ginsburg. *Object and Spreadsheet Histories*, chapter 12 in [17].

[13] Eric N. Hanson. The design and implementation of the Ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, 8(1), 1996.

[14] L. Edwin McKenzie Jr. and Richard T. Snodgrass. Evaluation of relational algebras incorporating the time dimension in databases. *ACM Computing Surveys*, 23(4):501-543, December 1991.

[15] Shamkant B. Navathe and Rafi Ahmed. *Temporal Extensions to the Relational Model and SQL*, chapter 4 in [17].

[16] A. Prasad Sistla and Ouri Wolfson. Temporal triggers in active databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):471-486, 1995.

[17] A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings, Redwood City, CA, USA, 1993.

[18] Jennifer Widom. A denotational semantics for the Starburst production rule language. *SIGMOD Record*, 21(3):4-9, 1992.

[19] Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, CA, USA, 1995.