

Parallel Event Detection in Active Database Systems: The Heart of the Matter

Ulrike Jaeger

Johann K. Obermaier

Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin
{jaeger, obermaier}@dbis.informatik.hu-berlin.de

Abstract

This paper proposes a strategy for parallel composite event detection in Active database systems (DBS). Up to now, the detection is sequential and totally synchronized, and thus preserves the timely order of events during the detection process. However, in distributed and extensible applications events may occur simultaneously in parallel unsynchronized streams. In order to adapt composite event detection to those new requirements we relax the timely order of events to a partial order and process parallel event streams. As a consequence, composite event detection must deal with unsynchronized and parallel event compositions. This paper investigates several parallel event detection strategies. We adapt techniques and optimizations from query execution in Relational DBS. However, queries and composite events differ fundamentally. The semantics of query execution is set oriented, whereas composite event detection bases on open streams of events. Cardinality and frequency of incoming events is unpredictable; timely order is essential. Parallel execution destroys the timely order of events and compositions. Our approach introduces a hybrid parallelization strategy for composite event detection in Active DBS that respects the timely order of events.

1 Introduction

We extend the concepts of Active DBS in order to meet the requirements of event driven, long lived, distributed, and partially connected applications. Those applications are systems of loosely coupled participants which coordinate and synchronize by exchange of event information. Participants can be software and hardware devices, as well as persons, who use a context information interface. The behavior of all participants is non-deterministic. A producer of event information is often unaware of possible consumers. The Consumption of event information depends on many parameters, e.g. it varies with the connectivity of the system. As a consequence, the relevance of event information is not static and must be expressed dynamically by explicitly parameterized requests for event information. The number of dynamic requests may be considerably large and requires efficient evaluation. Furthermore, the number of events and possible combinations of events is growing exponentially. If the event detection is not restricted to small time intervals, compositions for many requests lead to a considerable amount of data.

Up to now, Active DBS perform event detection sequentially. The existing execution models guarantee a timely ordered detection process, but they become inefficient if many events and event compositions are involved. We therefore investigate methods to parallelize the detection process. As a drawback, parallel execution destroys the timely order of events and compositions, which is a problem for time-sensitive combinations like sequences, for example.

The remainder of this paper is organized as follows: Section 2 sketches the state of the art in Active DBS with an emphasis on composite event detection by operator graphs. Section 3 discusses parallelization techniques developed for query execution in Relational DBS. We illustrate the different techniques by examples for detecting composite events. We discuss the special problems that arise in composite event detection if we naively adapt parallel query execution strategies. As a solution, Section 4 introduces our hybrid parallelization strategy for composite event detection. Section 5 discusses related research. Section 6 contains our conclusion and future work.

2 Sequential Composite Event Detection

Active DBS extend the regular DBS functionality by *Event-Condition-Action* rules, called *ECA* rules [6]. An *event* represents the successful execution of some operation within the database system or application. The *condition* tests the context state when the event occurs. The *action* is executed, when the condition evaluates to true. The application defines and produces *atomic events*, and requests for atomic as well as complex *composite events*.

The event detector component in Active DBS collects and distributes atomic events, and detects event compositions.

2.1 Basic Concepts

An *event* is a “happening of interest” [11]. Events occur repeatedly, therefore those occurrences are instances of a given *event type*. Throughout this paper we use the term *event* for an event instance or an occurrence.

Atomic Events

Events are persistent and ordered by global time stamps. The Active DBS attaches a time stamp to each incoming event. The time stamps are isomorphic to \mathcal{N} . In contrast to most Active DBS approaches, in this paper we also deal with simultaneous events.

Atomic event types are denoted by capital letters. Instances of an event type are represented by tuples, having a set of attributes like type, time stamp and others. In this paper we focus on type and time information only. An atomic event is a tuple ($\langle \text{type name} \rangle, \langle \text{time stamp} \rangle$). For better readability we omit the tuple notation and show an atomic event as an aggregate $\langle \text{type name} \rangle. \langle \text{time stamp} \rangle$. For example, $A.4$ is an instance of type A , having the time stamp 4 . Throughout this paper we use capital letters for type names as well as for the set of instances of those types.

Composite Events

Atomic event types are the alphabet of the event language. A composite event is a combination of constituent events which are combined and detected by the Active DBS. Languages for event compositions provide a variety of operators [3,18,25].

We distinguish three classes of operators: *constructors*, *collectors* and *selectors* of events. Constructors combine events from different sources to form new result tuples. Collectors collect events from different sources and merge them without constructing a new event. Selectors receive events from one source and select a certain subset without constructing a new event.

In this paper, we focus on a common subset of event language operators as examples for each operator class:

- *BEFORE* (A, B): instances of A are composed with instances of B if $a \in A$ happens before $b \in B$, i.e. *time stamp of a < time stamp of b* . The operator is a constructor for triples $(a, b, t) \in A \times B \times TIME$, where t is the time stamp of b .
- *AND* (A, B): instances of both A and B are composed, no matter what timely order. The operator is a constructor for triples $(a, b, t) \in A \times B \times TIME$, where t is the time stamp of the most recent of the two constituent events.
- *OR* (A, B): instances of either A or B , no matter what timely order. The operator is a collector of events from $A \cup B$. The result of *OR* is a heterogeneous set.
- *FIRST* (A): the oldest instance of A . The operator is a selector of events for a single $a \in A$, with *time stamp of $a \leq$ time stamp of a' , $\forall a' \in A$* .
- *LAST* (A): the most recent instance of A . The operator is a selector of events for a single $a \in A$, with *time stamp of $a \geq$ time stamp of a' , $\forall a' \in A$* .

Composition Semantics

The detection operates on sets of constituent events and produces sets of compositions. SENTINEL was the first Active DBS to provide a set of explicit language concepts to define which subset of possible combinations is required [5,15]. The semantics of event composition is described by so called “consumption modes”. The composition semantics determines the behavior of constructor operators, like *BEFORE* and *AND*. There are many variations of composition semantics (cf. [1,15]). In this paper, we refer to two: The one is the most general consumption semantics, which we call *ALL*. The other is the most commonly used semantics in Active DBS, called *CHRONICLE* [15].

Constructors with the *ALL* semantics are based on the Cartesian product of constituent events. Combinations are constructed according to the operator semantics. For example, *ALL* (*AND* (A, B)) results in a set S with: $|S| = |A| * |B|$ and $S \subseteq A \times B \times TIME$.

Constructors with the *CHRONICLE* semantics produce a subset of the result using the *ALL* semantic: combination of events also means their consumption. If an event is combined with others to form a composite event, it cannot be not reused for other composite events. *CHRONICLE* describes the stream based behavior of most of the event detection implementations in Active DBS. For example, input events for the *AND* operator are combined and consumed in fist-in-first-out (*fifo*) order.

2.2 Execution Model

An Active DBS creates composite event detectors that automatically collect and combine constituent events of the event composition. In this paper, we discuss the operator graph approach as used in SENTINEL [5,15], REACH [2], ADL [1], SMILE [13], and others [18,25]. Finite automata in ODE [11,12], and modified colored Petri nets in SAMOS [8,9,10] are variations of the detection technique based on operator graphs. All approaches are sequential. Event detection is centralized.

During runtime, the Active DBS receives atomic events and collects those in a timely ordered history. For example, a history of event instances of atomic types A, B, C, D is: $\langle A.1, B.2, C.3, D.4, D.5, C.6, A.7 \rangle$.

For composite events the detection process advances step by step with each constituent event. The composite event detector accepts events in a totally ordered stream. As soon as an event occurs, the detection process proceeds as far as possible. The most recent constituent event immediately causes the detector to signal the event composition.

A detector graph is represented by a set of nodes and edges. An *edge* indicates a stream of totally ordered entries. Entries are appended at the end of the stream and received from the head. A *node* is

either a leaf, a root, or an operator node. A detection graph forms a tree.

A leaf node is labelled by an event type. It represents the perception of constituent events of that type. It has one input stream for the required constituent event type and one output stream. A leaf performs selections on input streams, thus it is a selector operator.

An operator node represents an operator of the event language. It receives entries from a set of input streams and composes new result entries according to the operator definition. Operators use internal buffers for operands. The result composition is sent to the output stream. The output stream is input stream for another node.

The root node has one input stream and a set of output streams. The root prepares each entry for output to consumers. The root sends the result to the application and other consumers of the composition. A root is a selector operator.

Several detection trees form a forest, if they have common subexpressions. For example, if an event composition X is input to another event composition Y , the root for X is directly connected to the according leaf in Y .

For the sequential synchronized execution of the detector graph, we assume that new constituent events are sent to the leaf node synchronously. For each input entry the operator tree performs all possible operators, and produces results possibly up to the root. New input is accepted by the tree only if no further construction of results is possible. This strategy synchronizes event composition and guarantees that no event can overtake previous events during the detection process. For the *ALL* semantics constructor operators like *AND* and *BEFORE* have to buffer their input streams and reuse them for new combinations. For the *CHRONICLE* semantics the operands are not buffered. Input streams are received, combined, and consumed.

2.3 Example

Figure 1 shows the detection tree for the composition $(OR(A, BEFORE(B, AND(C, D))))$.

Table 1 shows the execution on the history $\langle A.1, B.2, C.3, D.4, D.5, C.6, A.7 \rangle$ using the *ALL* semantics. Events in the history have time stamps 1 to 7, attached by the event detector component.

In Table 1 the first column counts time ticks in order to show the detection process. In our model, each operator performs its work on a single entry within one tick. The other columns show the result tuple for each operator at the time when it is produced. We show the language operators and omit the root operator.

Table 2 shows the execution of the same detection tree using the *CHRONICLE* semantics. Here $B.2$, $C.3$, and $D.4$ are consumed in combinations. We show the deletion from the operator's buffers by shadowing the field at consumption time. Entries above the shadowed field are deleted. For example, $B.2$ is deleted at composition time tick 8. With consumption, $C.6$ and $D.5$ do not lead to new complete events.

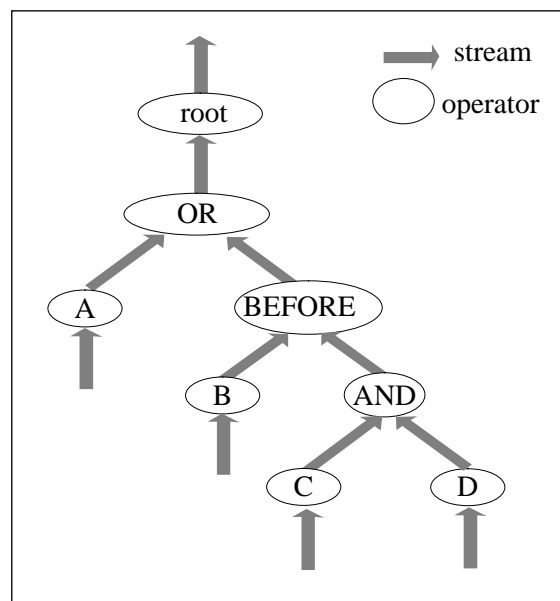


Figure 1: detection tree for $OR(A, BEFORE(B, AND(C, D)))$

<i>ticks</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>AND</i>	<i>BEFORE</i>	<i>OR</i>
1	<i>A.1</i>						
2							<i>A.1</i>
3		<i>B.2</i>					
4			<i>C.3</i>				
5				<i>D.4</i>			
6					<i>(C.3, D.4, 4)</i>		
7						<i>(B.2, (C.3, D.4, 4), 4)</i>	
8							<i>(B.2, (C.3, D.4, 4) 4)</i>
9				<i>D.5</i>			
10					<i>(C.3, D.5, 5)</i>		
11						<i>(B.2, (C.3, D.5, 5), 5)</i>	
12							<i>(B.2, (C.3, D.5, 5), 5)</i>
13			<i>C.6</i>				
14					<i>(C.6, D.4, 4)</i>		
15						<i>(B.2, (C.6, D.4, 6), 6)</i>	
16							<i>(B.2, (C.6, D.4, 6), 6)</i>
17					<i>(C.6, D.5, 6)</i>		
18						<i>(B.2, (C.6, D.5, 6), 6)</i>	
19							<i>(B.2, (C.6, D.5, 6), 6)</i>
20	<i>A.7</i>						
21							<i>A.7</i>

Table 1: Sequential example for *ALL (OR (A, BEFORE (B, AND (C, D))))*

<i>ticks</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>AND</i>	<i>BEFORE</i>	<i>OR</i>
1	<i>A.1</i>						
2	■						<i>A.1</i>
3		<i>B.2</i>					
4			<i>C.3</i>				
5				<i>D.4</i>			
6			■	■	<i>(C.3, D.4, 4)</i>		
7		■				■	<i>(B.2, (C.3, D.4, 4), 4)</i>
8						■	<i>(B.2, (C.3, D.4, 4) 4)</i>
9				<i>D.5</i>			
10			<i>C.6</i>				
11			■		<i>(C.6, D.5, 6)</i>		
12	<i>A.7</i>						
13	■						<i>A.7</i>

Table 2: Sequential example for *CHRONICLE (OR (A, BEFORE (B, AND (C, D))))*

2.4 Properties

The example shows the effects of synchronization:

- Synchronized sequential event detection is *time order preserving*.

Each row shows one result, meaning: each operator execution blocks other operators. In each col-

um the results are timely ordered. The execution does not destroy the timely order, due to synchronization. For example, in Table 2 the event *A.7* is accepted only after *C.6* has caused results up to the *OR* operator. *A.7* cannot overtake predecessors which would corrupt the timely order of the result tuples.

- Synchronized sequential event detection is *not efficient*.

Since only one operator can be active at a time, all others are idle. The small example shows a tendency that can be generalized for real world applications where many events occur. The *ALL* semantics causes exponential increase of intermediate results in the tree. The tree is blocked until all results are computed. New events will be accepted with increasing delays.

Therefore we looked for ways to speed up the composite event detection. Few application examples require big trees. As shown in [22], event compositions rarely exceed a depth of 5. But the number of events and intermediate result explodes exponentially. Especially in applications with many active detection trees or *ALL* semantics the sequential execution leads to considerable delays in composite event detection.

3 Problems of Parallel Event Detection

Parallelization is especially useful for operator trees as shown in 2.2. Stream based operators are well suited for parallel execution. We could relax the synchronization: while operator nodes compute results, the tree could accept new events and compute result sets in parallel to produce the complete set of final results much faster.

3.1 Parallelization Strategies

For parallelization we classify operators into three categories:

- *Pipelining operators without buffers*: the operators receive one or more streams of data entries, perform a computation on each single entry, and immediately produce an output entry. Example: root operator.
- *Pipelining operators with buffers*: the operators receive typically more than one stream of input entries, perform a computation on combinations of entries, and produce a number of sets of output entries. Those operators need buffers to collect entries of operands. In general, all constructor operators need buffers. Example: *BEFORE*.
- *Blocking operators*: the operators receive one or more streams of data entries, perform a computation on combinations of entries only if all operands are complete, and produce output entries. Those operators need buffers to collect entries of operands and will not produce any results before the last input entry has been sent. Example: *LAST*.

Two important parameters determine decisions to find the optimal parallelization strategy:

- The *workload* of a process, i.e. the amount of data it has to process, and the cost of processing the operator itself.
- The *communication cost* between processes, which is a consequence of the underlying architecture. Since we assume a shared everything architecture, we do not consider communication cost in this paper.

In general, operator trees can be executed in parallel by three different strategies:

- *Inter-tree-parallelism*: each detector is executed as a single process, all detectors run in concurrent processes.
- *Inter-operator-parallelism*: each operator in a tree is executed in a single specialized process.
- *Intra-operator-parallelism*: each operator is executed by a set of concurrent processes, ideally one for each data entry.

All three parallelization strategies can be combined. Inter-tree-parallelization alone leads to a heavy workload for each process, since a single process evaluates all data entries from leaf to root in a single sequential program. Both inter- and intra-operator-parallelism add to a much finer granularity and diminish the workload for each process such that we achieve a much better load balancing between processes.

Next, we investigate inter- and intra-operator parallelism and apply it to our language operators for examples. We call the first *pipelining parallelism strategy (PPS)*, the latter *universal parallelism strategy (UPS)*.

Pipelining Parallelism Strategy (PPS)

Each operator is performed by a specialized process. Operators are connected by sequential streams, see Figure 2.

After a start-up delay all operators receive and produce entries concurrently. For the three operator categories execution works as follows:

- Pipelining operators without buffers receive an entry, process it and produce a new result entry to the output stream. For example, the root receives an entry from either input stream, adds some context information and immediately sends it to the output stream.
- Pipelining operator with buffers receive an entry. For example, *BEFORE* searches the buffers for entries with appropriate time stamps, and combines pairs. For each entry there may be a set of matching pairs. That set is sent to the output stream, one at a time. Then the entry is added to the corresponding buffer.
- Blocking operators collect all incoming entries until the stream terminates. Then the operator is executed, and a single set of results is sent to the output stream, one at a time. Blocking operators require finite input streams. For example, the *LAST* operator is a blocking operator that is undefined when the input stream is infinite.

However, PPS has a serious load balancing problem: while the workload for each operator increases towards the root, the grade of parallelism decreases. The root is a bottleneck with heavy workload.

Universal Parallelism Strategy (UPS)

UPS adapts a strategy proposed for relational DBS to overcome the above load balancing problems for query optimization [17]. In the conventional parallelization paradigm operators are interpreted as processes, and stream entries as passive data. In UPS we interpret each stream entry as a process, and the operator tree as passive data. UPS requires a shared everything architecture. The detector tree information is stored in shared memory, and globally known to each process. For the model, each stream entry is a process. It reads the universal tree information and performs all operations from leaf to root node, see Figure 3. Of course, a realistic implementation cannot evaluate each event as a single process, but requires a suitable organization of processes and resources as proposed in [17]. For this paper, we do not discuss resource organization but the general principle of PPS. For the three operator categories execution works as follows:

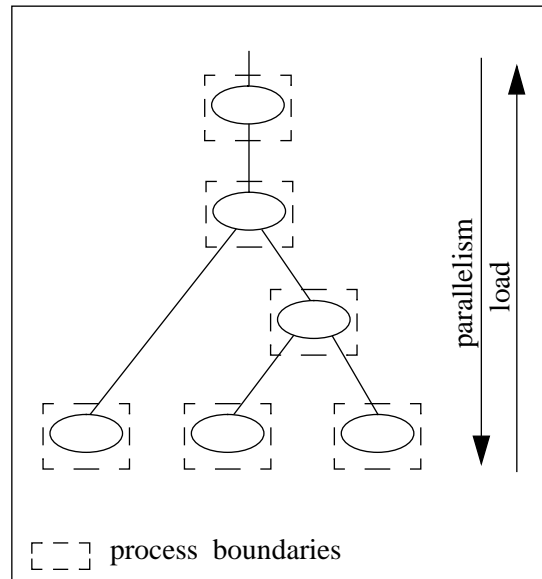


Figure 2: PPS

- If a process performs a pipelining operator without buffers, the process transforms the event information and resumes the program. For example, if an event performs the root operator, it attaches context information to its body and enters the output stream.
- If a process performs a pipelining operator, it creates a data entry and appends it to the according buffer. Next, it searches the opposite buffer for partners, creates new processes for each matching pair, and terminates. As an example, *AND* combines all entries in one buffer to each new entry of the opposite operand.
- If a process performs a blocking operator, it creates a passive data entry in the buffer. Only the last entry of a stream remains an active process and performs the operation on all data entries in the buffer. Of course, that operator will produce results only if the input streams are finite.

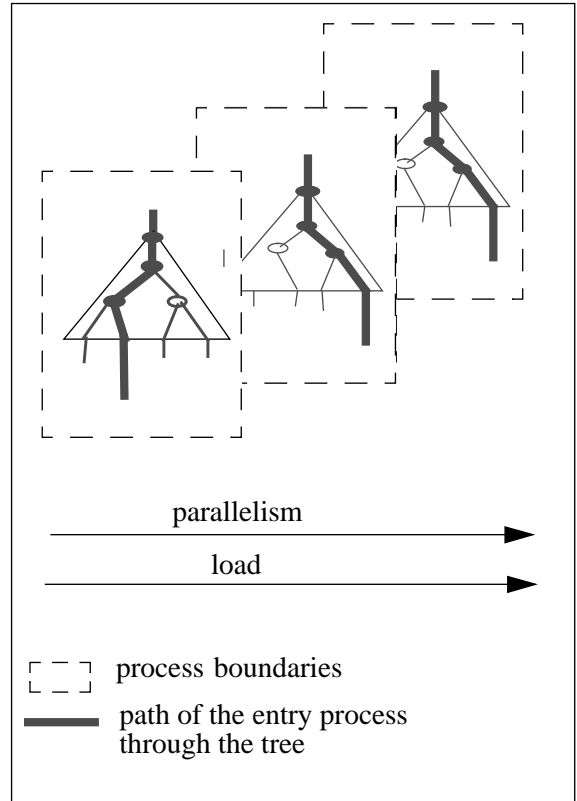


Figure 3: UPS

UPS achieves both pipelining and intra-operator parallelism, because each entry as a process performs all operators and between operators the streams are used in a pipelining manner. The fine granularity overcomes the problems of the PPS, because no operator is a bottleneck.

3.2 Example

We use the example in 2.3 for *ALL (OR (A, BEFORE (B, AND (C, D))))*. Again, we assume that each operation needs one tick for each entry. The events enter the tree in parallel. We could send all events at the same time, but in order to have a more natural behavior, our example schedule is partially ordered.

First, let us look at the PPS. Table 3 shows the results for each tick. After a certain start-up delay all operators receive and produce results in parallel. The parallel execution is considerably faster than the sequential execution with 21 ticks.

<i>ticks</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>AND</i>	<i>BEFORE</i>	<i>OR</i>
1	<i>A.1</i>	<i>B.2</i>					
2			<i>C.3</i>	<i>D.4</i>			<i>A.1</i>
3	<i>A.7</i>		<i>C.6</i>	<i>D.5</i>	<i>(C.3, D.4, 4)</i>		
4					<i>(C.3, D.5, 5)</i>	<i>(B.2, (C.3, D.4, 4), 4)</i>	<i>A.7</i>
5					<i>(C.6, D.4, 6)</i>	<i>(B.2, (C.3, D.5, 5), 5)</i>	<i>(B.2, (C.3, D.4, 4), 4)</i>
6					<i>(C.6, D.5, 6)</i>	<i>(B.2, (C.6, D.4, 6), 6)</i>	<i>(B.2, (C.3, D.5, 5), 5)</i>
7						<i>(B.2, (C.6, D.5, 6), 6)</i>	<i>(B.2, (C.6, D.4, 6), 6)</i>
8							<i>(B.2, (C.6, D.5, 6), 6)</i>

Table 3: PPS example for *ALL (OR (A, BEFORE (B, AND (C, D))))*

For the UPS example we use the same scenario. Each event is a process. Table 4 shows the content and position of an event process. For example, the process for *A.1* passes the leaf operator, and next per-

<i>ticks</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>AND</i>	<i>BEFORE</i>	<i>OR</i>
1	<i>A.1</i>	<i>B.2</i>					
2			<i>C.3</i>	<i>D.4</i>			<i>A.1</i>
3	<i>A.7</i>		<i>C.6</i>	<i>D.5</i>	$(C.3, D.4, 4)$		
4					$\{(C.3, D.5, 5),$ $(C.6, D.4, 6),$ $(C.6, D.5, 6)\}$	$(B.2, (C.3, D.4, 4), 4)$	<i>A.7</i>
5						$\{(B.2, (C.6, D.4, 6), 6),$ $(B.2, (C.3, D.5, 5), 5),$ $(B.2, (C.6, D.5, 6), 6)\}$	$(B.2, (C.3, D.4, 4), 4)$
6							$\{(B.2, (C.6, D.4, 6), 6),$ $(B.2, (C.6, D.5, 6), 6),$ $(B.2, (C.3, D.5, 5), 5)\}$

Table 4: UPS example for *ALL (OR (A, BEFORE (B, AND (C, D))))*

forms the *OR* operator, which takes one tick. It is now in the output stream of the *OR* operator, therefore shown as “result”. The result of *AND* at tick 4 is a set of events, all computed in parallel. That set proceeds further in parallel. That is why UPS needs less time than PPS which proceeds one tuple at a time. Especially for the *ALL* semantics, UPS will lead to substantial performance improvements and is a tempting technique.

3.3 Properties

Parallelization overcomes the synchronization delays of sequential execution. Unfortunately, parallel computation may corrupt the timely order of events.

PPS fills and processes streams one tuple at a time. As a consequence, the detection process along one path in the tree is order preserving. However, streams among each other are not synchronized. Therefore binary operators may receive entries of one stream that overtook entries of the other. For example, *A.7* arrives at the *OR* operator ahead of others with older time stamps (see Table 3, tick 4). We observe two reasons for the order destroying effect:

- The structure of the operator tree is unbalanced, i.e. the path lengths differ significantly. We do not assume that processing takes no time, therefore each operator along a path adds to the computation delay. Entries of short subtrees can overtake entries of deep subtrees.
- The entry load is unbalanced. Some constructor operators receive many entries and have to compute more combinations than others. The delay allows other streams to overtake.

In UPS both streams and operators are processed in parallel. Any given order will be destroyed by the execution. For example, the problem of *A.7* overtaking younger event compositions is due to the pipelining effects of the UPS (see Table 4, tick 4). In addition, operators are passed in parallel by a number of result processes. For example, events *C.3*, *D.4*, *D.5*, and *C.6* perform the *AND* operation and all following operators in parallel. Results from this set resume in arbitrary order (see Table 4, ticks 4 to 6).

In Relational DBS this behavior is not a problem. The semantics of query execution is set oriented and the base relations are complete. In contrast, composite event detection is based on open streams of events. Cardinality and frequency of incoming events is unpredictable and timely order is essential.

Therefore we cannot simply adapt parallelism for event detection, but we have to find a strategy that respects the timely order of events.

4 Hybrid Parallel Event Detection

The *ALL* semantics for composite event detection is based on sets and produces sets of combinations. A special order of results is not required. Therefore UPS is a suitable parallelization strategy. The resulting compositions will be produced in arbitrary order, but very quickly.

In composite event detection with *CHRONICLE* semantics the timely order of events is essential. The operands have to be sorted before producing results. Sorting might consume any performance improvement of the parallel execution. We will first introduce the problems of parallel execution of *CHRONICLE* detection trees, then introduce our parallel execution model that overcomes those problems.

4.1 Problems of Parallel *CHRONICLE* Execution

The *CHRONICLE* semantics for composite event detection is based on the consumption of constituent events in *fifo* order. Results are ordered streams of compositions.

UPS is not suitable for *CHRONICLE*, because the timely order of streams will be corrupted. All operators would have to sort input streams, thus becoming blocking operators. If the input streams are infinite, the tree will never produce results.

PPS is at least order preserving along paths in the detector tree. Since leaf input streams are timely ordered, we can directly apply PPS for parts of the detection process. Unary operators (*FIRST*, *LAST*, roots and leaves) in PPS are time preserving. But binary operators are confronted with two problems in PPS:

- Input streams are not synchronized among each other. We can not naively compose heads of the input streams as in the sequential execution in *fifo* order.
- Input streams for collector operators are unbalanced. Skew in event types can delay the execution unnecessarily: one operand is producing frequent results while the other does not. For example, the collector operator *OR* has to wait for entries from the silent operand in order to decide which events can be sent to the output stream. The operator cannot pipe incoming events as they arrive because the silence of one operator may be due to unbalanced subtrees and older entries may arrive after an unpredictable delay.

For example: Imagine a history $\langle B.1, C.2, \langle \text{an exclusive sequence of } 500 a \in A \rangle, D.503 \rangle$ for *CHRONICLE* (*OR* (*A*, *BEFORE* (*B*, *AND* (*C*, *D*))). All instances of *A* have to wait for the first instance of *D* before the *OR* operator can produce results.

4.2 PPS with Heartbeat

As stated before, unary operators are processed by PPS without any further extensions to the strategy. In order to produce sorted results for binary operators, we have to collect and sort entries in buffers before producing new results after comparing the time stamps. We propose sort-merge joins instead of naive *fifo* construction for constructor operators like *BEFORE* and *AND*.

In order to speed up production of results in data skew situations, we introduce *heartbeats*. A heartbeat is a special event that is produced globally and frequently. It bears a time stamp as any other incoming event and is of type *H*. The heartbeat type is a subtype of all other types. Copies of a heartbeat are sent to each leaf of the subtree of *OR* operators. The operators accept heartbeats and send them to their output streams. Since streams are timely ordered, the heartbeat *H.t* with time stamp *t* indicates that all entries in that stream preceding *H.t* are older or equal than *t*, while all entries following *H.t* are

younger or equal than t . As soon as an operator has received identical heartbeats from its input streams, it merges all entries in its buffers and sends them to the output stream. The buffers are flushed. The heartbeat is sent as well. As soon as both operands deliver entries, the heartbeat is no longer essential. The buffers can be merged and sent directly.

Example: We want to detect compositions $CHRONICLE (OR (A, B))$ in a history of atomic events: $\langle A.1, A.2, A.5, A.8, A.11, A.13, A.14, A.15, B.16, \dots \rangle$. We omit instances of other types, therefore some time stamps are not shown. Without heartbeat, all instances of A have to wait for the first instance of B . In order to speed up the OR execution, we insert two global heartbeats to the history: $\langle A.1, A.2, A.5, \mathbf{H.6}, A.8, A.11, A.13, A.14, A.15, B.16, \mathbf{H.16}, \dots \rangle$.

Table 5 shows a section of the execution: Both leaves eventually receive and propagate $H.6$, see ticks 4 and 6. Leaf B sends the heartbeat as its first entry, since $B.16$ has not yet occurred. As soon as OR has received two identical heartbeats from each of its input streams, it merges all buffered entries into the output stream. Since the right operand buffer contains no entries of type B , it sends the buffered section of $A.1, A.2$, and $A.5$, followed by the heart beat, see ticks 7 to 10. During the following period OR receives and buffers instances from both A and B before the next heartbeat $H.16$. The buffers are merged without waiting for the heartbeats, and $B.16$ is sent along with instances of A , see ticks 12 to 16.

The example illustrates that collector operators like OR now produce events at discrete heartbeats. The frequency of heartbeats depends on elapsed time and the number of events already arrived at the buffer. The frequency of heartbeats is adapted to the frequency of events. The detector may dynamically increase the frequency if its buffers overflow, or decrease if the operators receive too many heartbeats. We produce more than one global heartbeat for all detectors and use individual heartbeats for each disjoint OR subtree in the forest. We can adjust the heartbeat frequency to the behavior of individual subtree streams.

ticks	A	B	OR
1	A.1		
2	A.2		
3	A.5		
4	H.6		
5	A.8		
6	A.11	H.6	
7			A.1
8	A.13		A.2
9	A.14		A.5
10	A.15		H.6
11	H.16	B.16	
12		H.16	A.13
13			A.14
14			A.15
15			B.16
16			H.16
17

Table 5: Heartbeat example

4.3 Hybrid Parallelism in One Detector

The above solutions for different operator classes are efficient if they can be combined in detection trees and forests. The solutions are not completely orthogonal, but combinations are possible without negative impact.

Trees with ALL semantics are executed by UPS . Trees with the $CHRONICLE$ semantics are executed by PPS with heartbeat. In the case of common subexpressions, trees combine ALL and $CHRONICLE$, we have to distinguish which semantics overrules which. Given a composition Y based on composition X , we combine the strategies as shown in Table 6.

For the $CHRONICLE$ semantics, the operators need different support to guarantee timely ordered output streams. Again, the combination is possible without negative impact:

- Selector operators have no buffers and no further impact on other operators.
- Constructor operators have local buffers for sorting and merging without further impact on other operators.
- Collector operators have local buffers for merging and need heartbeats with little impact on other operators. Subtree leaves of collectors produce heartbeat entries which enter each operator of the

	<i>ALL (Y)</i>	<i>CHRONICLE (Y)</i>
<i>ALL (X)</i>	<i>Y and X: UPS</i>	<i>Y and X: PPS & heartbeat</i>
<i>CHRONICLE (X)</i>	<i>Y: UPS X: PPS & heartbeat</i>	<i>Y and X: PPS & heartbeat</i>

Table 6: Parallel strategies for *ALL* and *CHRONICLE* combinations

subtree. In the subtree unary operators just receive and sent the heartbeat as any other entry. Binary operators perform sort-merge of the stream sections between two heartbeats, and send the upper bound heartbeat to the output stream.

The heartbeat limits buffer space and enables the detector to produce possible event compositions as early as possible. In contrast to sequential detection, which immediately produces the first composition instance, the heartbeat pulse might delay the first composition instance for at most one heartbeat. But PPS then will produce results much faster than a sequential execution.

5 Related Research

At first view the issue of complex event detection with operator trees is closely related to the processing of time-series in temporal data base systems (cf. [20,23]), or the processing of general sequence data (cf. [21]). In those areas the timely order of data is essential. But temporal queries process completely stored data. This correspond only to composite event detection in a persistent history of past events, whereas the immediate detection of new composition instances from open streams is a new issue here.

Parallel processing in Relational DBS has been a research topic for a long time [7,24]. Relational DBS are good candidates for performance improvements by parallelization because of the set-oriented nature of relations. The sequence-oriented second nature of time-series spoils set-oriented parallelization in temporal database systems. So, research for temporal databases is taking very first steps towards parallel processing [14,16]. The considered parallelization strategies are only feasible for stored data, and not for continuous processing as needed for composite event detection.

Most Active DBS approaches propose centralized, sequential detection of composite events. Schwiderski et al. [19] discuss time stamping and timely order of results in distributed composite event detection based on detector trees with the *CHRONICLE* semantics. The trees themselves are distributed across sites. The approach introduces two detection algorithms: a synchronized algorithm where operator nodes request for each input entry from other sites and an asynchronous algorithm where nodes accept input entries irrespectively of timely order. The synchronized algorithm enforces timely order of results, which may lead to unpredictable delays due to failure of sites. In that case, delays may completely block the operator. Entries are buffered and merged as in our PPS approach. Parallelization itself is not the scope of the paper, but the distributed detection process leads to concurrent evaluation. In contrast, our approach assumes centralized event detection. We propose a parallel strategy suitable for the *CHRONICLE* and the *ALL* semantics, respecting timely order of events, and use both PPS and UPS.

6 Conclusion and Future Work

Our hybrid approach for parallel event detection is a useful optimization for applications where many trees have to cope with frequent and unsynchronized events. The paper discusses the parallelization

with the help of a subset of event operator languages. The basic discussion of collectors, constructors, and selectors can be extended and applied to other Active DBS languages as well.

Our Prototype implementation currently is based on the parallel programming language C-LINDA [4], having a network of 6 SPARCstations with 8 processors in total. Both PPS and UPS are implemented for the discussed example operators. The next step is to automatically create dynamic individual heartbeats adjusted to a hybrid parallelization of arbitrary event compositions, where the *ALL* and the *CHRONICLE* semantics are combined.

Acknowledgments

The authors would like to thank Johann-Christoph Freytag for many useful comments on the presentation of the paper.

References

- [1] H. Behrends. *Specification of Event Driven Activities in Data Based Information Systems*. Ph.D. Thesis, University of Oldenburg, Germany, October 1995 (in German).
- [2] A. P. Buchmann, H. Branding, T. Kudrass, and J. Zimmermann. REACH: A REal-Time, ACtive and Heterogenous Mediator System. *Data Eng. Bull.*, 15(1-4), December 1992.
- [3] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proc. Int'l. Conf. on Data Eng.*, Taipei, Taiwan, March 1995.
- [4] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, (21)3, September 1989.
- [5] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases. Semantics, Contexts and Detection. In *Proc. Int'l. Conf. on Very Large Data Bases*, Santiago de Chile, Chile, September 1994.
- [6] U. Dayal, A. P. Buchmann, and D. R. McCarthy. Rules are Objects Too: A Knowledge Model For An Active, Object-Oriented Database System. In *Proc. Int'l. Workshop on Advances in Object-Oriented Database Sys.*, Volume 334 of *Lecture Notes in Computer Science*. Springer Verlag, 1988.
- [7] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6), June 1992.
- [8] S. Gatzju. *Events in an Active, Object-Oriented Database System*. Ph.D. Thesis, University of Zurich, Switzerland, Verlag Dr. Kovac, 1995.
- [9] S. Gatzju and K. R. Dittrich. SAMOS: An Active Object-Oriented Database System. *Data Eng. Bull.*, 15(1-4), December 1992.
- [10] S. Gatzju and K. R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *Proc. Research Issues on Data Eng.: Active Database Systems*, Houston, Texas, USA, February 1994.
- [11] N. H. Gehani, H. V. Jagadish and O. Shmueli. Event Specification in an Active Object-Oriented Database. In *Proc. ACM SIGMOD Int'l. Conf.*, San Diego, California, USA, June 1992.
- [12] N. H. Gehani, H. V. Jagadish and O. Shmueli. Composite Event Specification in Active Databases: Model and Implementation. In *Proc. Int'l. Conf. on Very Large Data Bases*, August 23-27, Vancouver, British Columbia, Canada, August 1992.
- [13] U. Jaeger. SMILE — A Framework for Lossless Situation Detection. In *Proc. Int'l. Workshop on Inf. Technologies and Sys.*, Nijenrode, Netherlands, December 1995.
- [14] S. Karimi, M. Bassiouni, and A. Orooji. Supporting Temporal Capabilities in a Multi-Computer

- Database System. In *Proc. Int'l. Conf. on Databases, Parallel Architectures, and Their Applications*. Miami Beach, Florida, USA, March 1990.
- [15] V. Krishnaprasad. *Event Detection for Supporting Active Capabilities in an OODB: Semantics, Architecture and Implementation*. Masters Thesis, University of Florida, Gainesville, Florida, USA, 1994.
- [16] T. Y. C. Leung and R. R. Muntz. Temporal Query Processing and Optimization in Multiprocessor Database Machines. In *Proc. Int'l. Conf. on Verly Large Data Bases*, Vancouver, British Columbia, Canada, August 1992.
- [17] S. Manegold, J. K. Obermaier, F. Waas, and J.-C. Freytag. Load Balancing in Shared Everything Environments. Technical Report HUB-IB-61, Humboldt-University of Berlin, Germany, May 1996.
- [18] N. Paton and H. Williams (Eds.). *Proc. Int'l. Workshops on Rules in Database Systems*, Edinburgh, Scotland, August/September 1993. Workshops in Computing, Springer 1994.
- [19] S. Schwiderski, A. Herbert, and K. Moody. Composite Events for Detecting Behavior Patterns in Distributed Environments. In *TAPOS Distributed Object Management*, July 1995.
- [20] A. Segev and A. Shoshani. Logical Modeling of Temporal Data. In *Proc. ACM SIGMOD Int'l. Conf.*, San Francisco, California, USA, May 1987.
- [21] P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proc. Int'l. Conf. on Verly Large Data Bases*, Bombay, India, September 1996.
- [22] E. Simon and A. Kotz-Dittrich. Promises and Realities of Active Databases. In *Proc. Int'l. Conf. on Verly Large Data Bases*, Zurich, Switzerland, September 1995.
- [23] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases — Theorie, Design, and Implementation*. Benjamin/Cummings, Redwood City, California, USA, 1993.
- [24] P. Valduriez. Parallel Database Systems: Open Problems and New Issues. *Distributed and Parallel Databases, An Int'l. Journal*, 1(2), 1993.
- [25] J. Widom and S. Chakravarthy (Eds.). *Proc. Int'l. Workshop on Research Issues in Data Engineering: Active Database Systems*, Houston, Texas, USA, February 1994.