HÖGSKOLAN SKÖVDE

Institutionen för kommunikation och information
Examensarbete i datavetenskap 25 p
D-nivå
Vårterminen 2005

Parameterized Event Monitoring

Priyadarshini Dande

HS-IKI-MD-05-007

Parameterized Event Monitoring

Submitted by Priyadarshini Dande to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the School of Humanities and Informatics.

2005-09-26

I hereby certify that all material in this dissertation which is not my own work has been identified and that no work is included for which a degree has already been conferred on me.

Signature: _____

ACKNOWLEDGEMENTS

This is very small attempt to express my gratitude to those witty brains, dexterous hands, helping hearts and above that all the industrious people, who gave their attention, patience and strength to keep myself going with the pulse of the thesis and to come up with this report today

I am highly grateful to my Supervisor Dr. Jonas Mellin without whose strong determination and motivation my thesis work is lifeless. Then I would like to extend my thanks to my examiner Prof. Sten Andler under whose surveillance, the thesis is ratified. I am especially very much bound to the program coordinator, Dr. Mikael Berndtsson who never ignored any of my doubts from the day I got admission.

Apart from these I like to extend my special vote of thanks to,
My opponent Sanny Gustavsson, whose questions revived some untouched percepts of subject and helped me to improve my thesis,
Robert Nilsson who spared no clause that interrupted my thesis work in my supervisors absence,
Parents,
Friends and classmates, and
Finally the STINT foundation whose uninterrupted financial support made my thesis a success today.

Parameterized Event Monitoring

Priyadarshini Dande

Abstract

Event monitoring has been employed in many applications such as network monitoring, active databases etc.; however, there is only an insignificant amount work done on parameterized event monitoring, a feature that is necessary in any real application. The aim of this work is to investigate solutions for parameterized event composition that is scalable and efficient; these solutions are refined from existing event monitoring algorithms. An algorithm for parameterized event composition is proposed and analysis on algorithmic time complexity is performed. In addition to this, experiments on the prototype Solicitor, a software component in DeeDS, along with simulated input of events are conducted in order to validate the theoretical model and the hypothesis that were made. The experiments support the theoretical model and suggest that it is possible to build an efficient and scalable parameterized event composition that is useful in real applications.

Key words: Event monitoring, parameterized event monitoring, parameterized event composition.

Contents

1 Introduction

Event monitoring is the process of collecting, analyzing and signaling events to the subscribers. The major part of event monitoring is event composition (i.e., analyzing events) and to understand this, consider an example where we need to detect an event E that occurs only if event E_1 is followed by an event E_2 and the event history contains a set of events: {E_1 at time stamp 1, E_1 at time stamp 2, E_2 at time stamp 3 and E_2 at time stamp 4}. Assume that events are analyzed or composed based on chronicle context, then the event history contains event occurrences: {E terminating at time stamp 3 (formed because E_1 at time stamp 1 is followed by E_2 at time stamp 3) and E terminating at time stamp 4 (formed because E_1 at time stamp 2 is followed by E_2 at time stamp 4)}. Suppose, E_1 at time stamp 1 and E_2 at time stamp 3 belong to two different applications then the detected event E at time stamp 3 is an incorrect composite event and this incorrect event detection can be prevented by using parameterized event monitoring. The research work in this paper deals with parameterized event monitoring.

Event monitoring based on parameterized event composition is called parameterized event monitoring. Parameterized event composition solves the problem of incorrect event composition and reduces the size of event monitor specification. The event composition solely based on event types and time stamps may result in incorrect event detection. If the composition of events is based on both the application identifiers in this case and event type and time stamp information then one can prevent this incorrect detection of events. This is due to that; parameterized event composition avoids pairing of events, E_1 at time stamp1 with E_2 at time stamp3 because the application identifiers are not equal. In parameterized event composition, the composition of events is performed based on key parameters, briefly, parameters that are specified to be used in event composition. Evaluation of an event expression depends upon the parameters of the contributing events because composite events are formed if and only if the contributing events have same parameter and value. This is one way of analyzing events where contributing events have same parameter and value (similar to an equi-join in relational database management systems (Ramakrishna, 2003)). Other type of join operations can also be considered, but equi-join is assumed to be the most efficient join and is considered in this thesis. To locate events having same parameters, an efficient indexing technique is required. In this thesis, we assume that it is feasible to construct an efficient and scalable event composition although additional computational steps are required for checking the parameter equivalence condition of the contributing events. An algorithm for parameterized event composition for binary operators is presented along with the algorithmic time complexity and experiments on parameterized event composition are conducted in order to study the affect of parameterized event composition on an event monitor.

In this report, section 2 provides information on event monitoring, its classification, structure, event composition and benefits of parameterized event monitoring. Section 3 describes the problem domain, motivation behind, objectives, hypothesis and assumptions and pre-requisites required for solving the problem. Architecture of parameterized event composition along with the algorithmic time complexity is discussed in section 4. Section 5 deals with the empirical study made on parameterized event composition. Comparison between parameterized event composition and non-parameterized event composition with respect to the factors such as efficiency and scalability is presented in section 6. Sections 7, 8 and 9 provide discussion, related work and conclusion of this research work.

2 Event Monitoring in Distributed Real-Time Systems

According to Burns & Wellings (2001), a r*eal-time system* reacts to stimuli in an environment in a timely fashion and a *distributed system* consists of multiple autonomous nodes (processing elements), cooperating for a common purpose. A *distributed real time system* can be viewed as a system of multiple number of nodes (real-time systems) interacting with each other and reacting to the stimuli in an environment in order to achieve a common goal. To achieve real-time constraints several nodes are distributed in contrast to a centralized system. This is because distribution of nodes becomes significant if there are any environmental (geographical) constraints (for e.g., in satellite communication), or to achieve fault tolerance through node redundancy such that if one node fails, the other serves or to have load balancing as one of the requirements for an application.
As addressed by, for example, Mellin (2004, p.1), stimuli in a real-time system environment can be viewed as events. In general, an *event* is defined to be an instantaneous, atomic (happens completely or not at all) occurrence of interest at a point in time (Chakravarthy et al., 1994). The primitive events are considered to be instantaneous and atomic but composite events are considered only to be atomic as defining composite events, as atomic, leads to incorrect results (Galton and Augusto (2001)). Events can be monitored by an event monitor. Event monitoring is used in several applications. For example, in active databases, the purpose of event monitoring is to optimize performance of the rule-processing and many approaches integrate event monitoring in rule processing. (e.g., Dayal et al. (1988), REACH (Buchmann et al., 1995)). In network monitoring, failures occurring in remote procedure calls are detected and the system is prevented from an error (Mok and Liu, 1997a). Similarly, a distributed intrusion detection system presented by Snapp et al. (1991) uses different forms of event monitors (e.g., LAN monitor, host monitor, etc) such that it can detect an intruder (unauthenticated user) accessing the network system.
By using event monitoring, one can detect failures, achieve observability and can have a control on several processes of a system. Although event monitoring provides these benefits, it affects the response times of a system as mentioned by Schroeder (1995) since the inserted code of event monitoring causes intrusion. This affects the performance of the system but event monitoring is acceptable provided there is no much variation shown in terms of computational costs incurred between the code with event monitoring and without event monitoring.
According to Mellin (2004), an *event monitor* provides a means for collecting, analyzing and signaling of event occurrences to subscribers. The structure of event monitoring is explained in section 2.1.

2.1 Structure of Event Monitoring

An overview of the event monitor based on Mellin (2004, p.13) is depicted in the Fig 2.1. Event monitoring collects events from the monitored object and stores them into a database called *filtered event log.* The filtered event log contains a subset of an event history without useless observations (e.g., observations that are too old to be meaningful for current or future decisions), whereas an *event history* contains all observations. The observations are analyzed and presented to a subscriber such as a user or a component. The subscriber performs actions based on the results obtained from the analysis. During analysis, the intermediary results are

written to the filtered event log for further analysis in future. In this thesis, event monitoring for on-line usage and a push protocol is considered. An *on-line* monitor executes data collection, event composition and action execution on the same resources as the tasks of the monitored object. The analysis phase (highlighted in the Fig 2.1) in the event monitor can be viewed as event composition and the details regarding event composition and parameter contexts are presented in the next section.

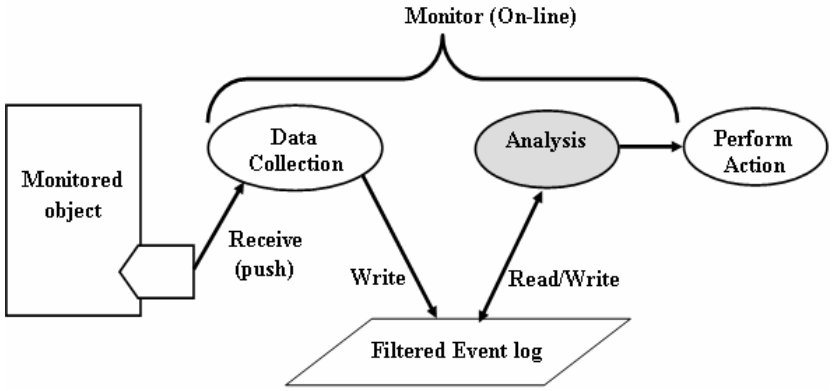

Figure 2.1: An overview of the event monitoring structure [Based on Mellin (2004)]

2.2 Event Composition and Event Contexts

The *event composition* is the process of analyzing event occurrences and assembling them into composite event occurrences. A similar concept of event composition is event detection where the composite events produced during event composition, forms as an input to the process of event detection where only the significant events that are of interest are detected. Event detection is available in active databases, for example, Snoop by Chakravarthy et al. (1994); SAMOS by Gatziu (1994); ODE by Gehani et al. (1993). In active databases, the patterns of database events are monitored such that whenever an event is detected the associated rule is triggered.

Event occurrences are categorized by event types. As defined by Mellin (2004, p. 40) an *event type* of an event occurrence have the same characteristic behavior and possibly be expressed in terms of other event occurrences. As in active databases, event types are either primitive or composite (Dayal et al., 1988; Chakravarthy et al, 1989; Chakravarthy and Mishra, 1994). A *primitive event* is a pre-defined event in the monitored object denoting that something important has occurred. The formation of a composite event is based on an event expression. An *event expression* is a pattern of events containing primitive or composite or a combination of both. This pattern defines how event occurrences of specified event types can be combined to form a composite event. Before we present the actual working mechanism of event composition, consider few basic formal definitions taken from Solicitor event specification language developed by Mellin (2004, p. 53).

Basic formal definitions:

Let G (generated set) represents event history and contains all asserted primitive and composite event occurrences. $\Gamma(E, [t, t'])$, is an event occurrence of event type E occurred during the interval [t, t'] where $t \leq t'$.

To understand the concept of event composition, consider event history:

Example 2.2:

G = {$\Gamma(E_1, [1, 1])$, $\Gamma(E_2, [2, 2])$, $\Gamma(E_2, [3, 3])$, $\Gamma(E_1, [4, 4])$, $\Gamma(E_1, [5, 5])$, $\Gamma(E_2, [6, 6])$} and $E_{12}=E_1; E_2$ is the event expression to be monitored where E_1 is the initiator and E_2 is the terminator. In a sequence operation (;), event E_1 is always followed by E_2. Initiator is the event which starts the event composition process and terminator ends the process.

If event composition is performed with this event history as the input then the filtered event log contains the following:

G = { $\Gamma(E_1, [1, 1])$, $\Gamma(E_2, [2, 2])$, $\Gamma(E_2, [3, 3])$, $\Gamma(E_1, [4, 4])$, $\Gamma(E_1, [5, 5])$, $\Gamma(E_2, [6, 6])$,$\Gamma(E_{12}, [1, 2])$, $\Gamma(E_{12}, [1, 3])$, $\Gamma(E_{12}, [1, 6])$, $\Gamma(E_{12}, [4, 6])$, $\Gamma(E_{12}, [5, 6])$}

This is depicted in the Fig 2.2. In this case, the algorithmic time complexity of event composition is conjectured to be exponential in terms of generating all composite event occurrences with respect to the number of event operators in an event expression. In addition to this, the analysis of event composition of a complex event expression is highly intractable (Berndtsson et al., 1999) and it is also difficult to understand the semantics of a complex event expression. Therefore, in order to achieve meaningful occurrences, parameter contexts, also known as event contexts (Mellin 2004), are addressed within Snoop by Chakravarthy et al. (1994, 1998). An *event context* defines what combination of contributing event occurrences are meaningful to combine into a composite event occurrence in a specific situation. Chakravarthy and Mishra (1994) named this unconstrained event context as general context and it is illustrated in the Fig 2.2. Since general context leads to non-scalable solutions Chakravarthy et al. (1994) has proposed four parameter contexts that are useful for wide range of applications. The description of four event contexts is illustrated below by considering the event history of example 2.2.

1. **Recent Context:** This context is useful for sensor applications (e.g., hospital monitoring, global position tracking) because the analysis phase considers latest event occurrences among several event occurrences of the same event type. Event composition based on recent context is shown in Fig 2.2 and on applying recent context, the filtered event log contains: {$\Gamma(E_{12}, [1, 2])$, $\Gamma(E_{12}, [1, 3])$ and $\Gamma(E_{12}, [5, 6])$}. Each new terminating event occurrence is combined with the most recent initiator to form a composite event. Event $\Gamma(E_1, [4, 4])$ is not selected because $\Gamma(E_1, [5, 5])$ is a more recent one.
2. **Chronicle Context:** Applications that exhibit causal dependency (e.g., between aborts, roll backs and other operations) are found under this category. In this context, the oldest initiator is paired with the oldest terminator for each event (i.e., in chronological order of event occurrence). The event composition based on chronicle context yields composite event occurrences: {$\Gamma(E_{12}, [1, 2])$ and $\Gamma(E_{12}, [4, 6])$}.Event $\Gamma(E_2, [3, 3])$ is invalidated by the sequence operator and cannot contribute to a future occurrence of composite event E_{12}.
3. **Continuous Context:** Trend analysis and forecasting applications (e.g., securities trading, stock market) where composite event detection along a moving time window needs to be supported. In this context terminator occurrence can be used together with several initiator occurrences to contribute to several composite event occurrences (one for each initiator occurrence). For the given history mentioned in example 2.2, the continuous context yields composite events: {$\Gamma(E_{12}, [1, 2])$, $\Gamma(E_{12}, [4, 6])$ and $\Gamma(E_{12}, [5, 6])$}. Event $\Gamma(E_2, [3, 3])$ is invalidated since there are no more

initiator occurrences left after the composition of a composite event $\Gamma(E_{12}, [1, 2])$ as shown in the Fig. 2.2.

4. **Cumulative Context:** This context is useful in applications where an event is terminated by a deadline event and all occurrences of constituent events are meaningful up to the occurrence of the deadline event (e.g., banking application). In this context, for each constituent event, all occurrences of the event are accumulated until the composite event is detected. For the given event history, cumulative context yields composite events: $\{\Gamma(E_{12}, [1, 2])$ and $\Gamma(E_{12}, [4, 6])\}$. Event $\Gamma(E_2, [3, 3])$ is invalidated since the initiator event $\Gamma(E_1, [1, 1])$ is consumed by the terminator event $\Gamma(E_2, [2, 2])$. Accumulation of events between the initiator and terminator takes place provided there are no intermediary terminator events in between the initiator and the terminator (deadline event). Event $\Gamma(E_2, [3, 3])$ is not considered because there is an intermediary event $\Gamma(E_2, [2, 2])$ between $\Gamma(E_1, [1, 1])$ and $\Gamma(E_2, [3, 3])$. Composite event $\Gamma(E_{12}, [4, 6])$ is formed by the contributing events: $\Gamma(E_2, [4, 4])$, $\Gamma(E_2, [5, 5])$ and $\Gamma(E_2, [6, 6])$.

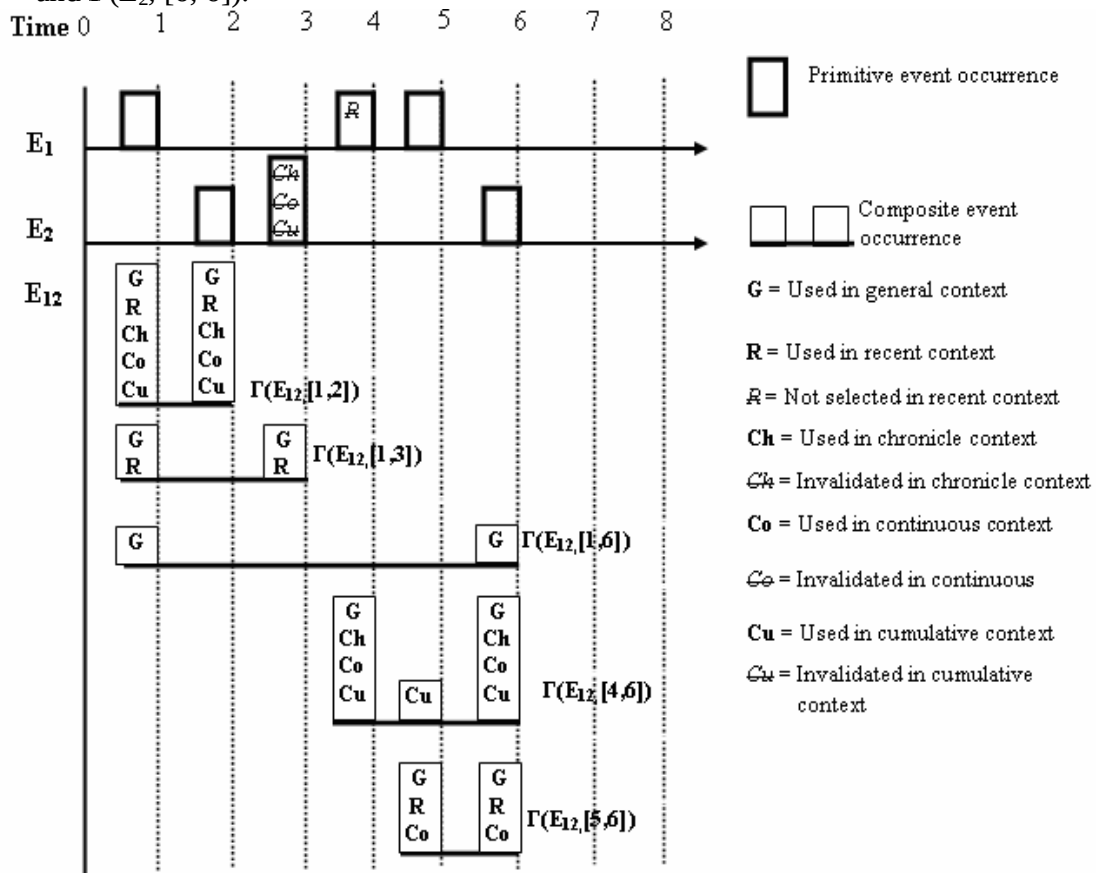

Figure 2.2: Depiction of Example 2.2 in general, recent, chronicle, continuous and cumulative event contexts

Thus, according to Mellin (2004) recent and continuous event contexts allow reusability of contributing event occurrences where the former (i.e., recent context) allows reuse of initiator and later (i.e., continuous context) allows reuse of terminator. Chronicle and cumulative contexts do not allow reusability and former allows a strict historical order during event composition. In conclusion, event composition based on event contexts

reduces the algorithmic time complexity from exponential (e.g., unrestricted context) to polynomial and is also scalable and tractable even in the case of complex event expressions. Some of these event contexts are implemented in several event specification languages such as Chronicle Recognition (Dousson et al., 1993), Snoop (Chakravarthy and Mishra, 1994), TCCS (Baekgaard and Godskesen, 1997), and Monitoring of Real-Time Logic (Mok and Liu, 1997a).

2.3 Parameterized Event Monitoring

The parameterized event monitoring differs from the concept of non-parameterized event monitoring in the case of event composition (i.e., in analysis phase as shown in the Fig 2.1). Event composition is based on key parameters. A key parameter is a parameter on which event composition is based (e.g., contributing event occurrences having equal parameters). Event composition can be classified into two types: non-parameterized and parameterized event composition. In non-parameterized event composition, analysis of event occurrences is based only on implicit key parameters of the event occurrences. Implicit key parameters include event type identifier, the time interval in which event has occurred and parameter context. Explicit key parameters are specified by the user[1].

The analysis of parameterized event composition involves both implicit and explicit key parameters, where a composite event is formed if and only if the explicit key parameters of the contributing event occurrences are same and also follow the rules of event composition. Due to this, the performance of parameterized event composition would be less than a non-parameterized event composition and also issues such as scalability and resource predictability are questionable. The need for parameterized event composition can be explained with an example.

Example 2.3.1: Consider library monitor introduced by Baekgaard and Godskesen, (1997) where the issued books are monitored such that they are returned with in a stipulated time (30 days). To validate the book borrowing mechanism, consider a simple composite event $E_{ca} = E_{borrow} ; E_{return}$ formed by a sequence operator where E_{ca} – correctly arrived composite event (i.e., book is arrived with out any delay), E_{borrow} – book borrow event and E_{return} – book return event. The event expression E_{borrow} followed by E_{return} is considered because E_{ca}, correctly arrived composite event is triggered when borrower borrows a book (i.e., when a primitive event E_{borrow} has occurred) and returns the book with out any delay (i.e., when a primitive event E_{return} has occurred). Assume event history $G = \{\Gamma(E_{borrow}, [1, 1]), \Gamma(E_{borrow}, [2, 2]), \Gamma(E_{return}, [3, 3]), \Gamma(E_{return}, [5, 5])\}$. If we perform event composition based on general context then the event history $G = \{\Gamma(E_{ca}, [1, 3]), \Gamma(E_{ca}, [1, 5]), \Gamma(E_{ca}, [2, 3]), \Gamma(E_{ca}, [2, 5])\}$ is obtained and is depicted in Fig 2.3.1. To restrict the composite events, assume event composition based on chronicle context is performed in order to control the combinatorial explosion of events. Only two composite events are generated (depicted in the Fig. 2.3.2) and event history G contains $\{\Gamma(E_{ca}, [1, 3]), \Gamma(E_{ca}, [2, 5])\}$. Now the problem is if E_{borrow} in the interval at [1, 1] is from borrower 1 and E_{return} in the interval at [3, 3] is from borrower 2 then the formed composite event, E_{ca} in the interval [1, 3] is an incorrect composite event which results in the case of non-parameterized event composition. This problem can be solved by parameterized event composition where external parameters such as book identifier and borrower identifier can be considered during the composition of events such

[1] Explicit non-key parameters are also specified by the user but they do not affect the process of event composition. They are just like non-key attributes of equi-join in relational database management systems.

that a parameter check is performed on the contributing events. Only the events having the same book identifier and borrower identifier are contributed to form a composite event. The problem behind non-parameterized event composition is identified by Mellin (2004, p. 56).

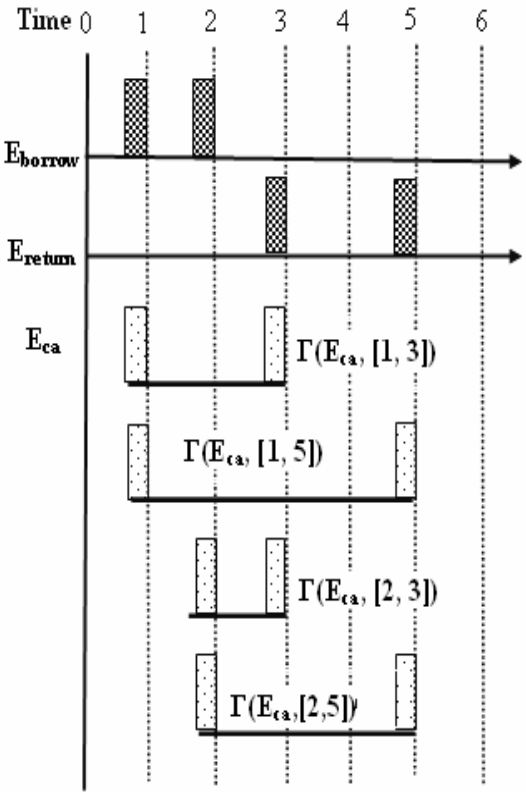

Figure 2.3.1:
Composition of $E_{ca} = E_{borrow}$; E_{return} in general event context

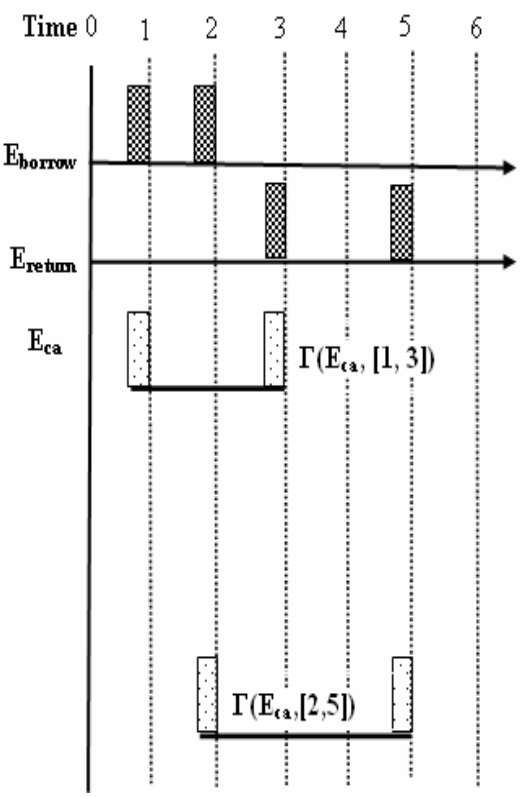

Figure 2.3.2:
Composition of $E_{ca} = E_{borrow}$; E_{return} in chronicle event context

Legend:

Primitive event occurrence

Composite event occurrence

This concept of parameterized event composition lacks in several event specification languages such as Chronicle Recognition (Dousson et al., 1993), Snoop (Chakravarthy et al., 1994), Monitoring of Real-Time Logic (Mok and Liu,1997a ; Mok and Liu,1997b). Parameterized event types are addressed in SAMOS (Gatziu and Dittrich, 1992; Gatziu et al., 1994), TCCS (Baekgaard and Godskesen, 1997) and implicitly mentioned in REACH (Buchmann et al., 1995). SAMOS an active database prototype based on Petri-nets supports parameterized event types having parameters as transaction identities and user identities. Baekgaard and Godskesen (1997) address parameterized event types in their specification language called TCCS. Buchmann et al. (1995) presented REACH, a layered architecture for OODBMS. It handles contributing event types stemming from different transactions in an appropriate way while performing event composition.

An event monitor specification for monitoring a library database is depicted in Fig 2.3.3. The primitive events *book1* to *bookN* denote books borrowed by students using their library

access card. The primitive event type *returnN* denotes that book is returned to the library. Two examples of behaviors are presented. The behavior *correct1* is signaled when there is a *book1* event occurrence followed by a *return1* event occurrence without any intermediate *delay* event occurrence. Any initiated event composition of *correct1* expires within 30 days. The *incorrect1* behavior is that a *book1* event occurrence is followed by a relative temporal event occurrence *book1+30 days* without any intermediate *return1* event occurrence. Both of these specifications are monitored by event monitoring task *taskOne* whose default event context is chronicle.

```
monitor LibraryMonitor{
   primitive{
      event book1(int librarycard).
      event book2(int librarycard).
      event book3(int librarycard).
        ⋮
      event bookN(int librarycard).
      event return1.
      event return2.
      event return3.
        ⋮
      event returnN.
      event delay.
    }//end of primitive event declarations

   task taskOne<context : chronicle>{
      event correct1(int librarycard) is
         N<expire:30 days>(book1,delay,return1).
      event incorrect1(int librarycard) is
         N(book1,return1, book1+ 30days).
     }//end of taskOne
    ….//repeat for each book/return event type

}//end of monitor
```

Figure 2.3.3: Event monitor specification example

This specification increases with the number of monitored objects (i.e., books in this example). In this case, the factor of scalability is a question because in many real-world applications, the number of non-parameterized event types (e.g., *book1,…,bookN*) emulating parameterization grows exponentially $O(k^n)$ where k is the key size and n is the number of key parameters. Therefore, it is not advisable to automatically generate event monitor specifications in the case of applications involving large key parameter domains. For small applications, this might be feasible but for large applications non-parameterization with dynamic generation of events for each combination of parameter and event type is not feasible.

The parameterized event monitor specification for library monitor is depicted in the Fig.2.3.4. The syntax for specifying this is adopted from Mellin (2004, p. 318). Here, book id (an identifier for book), a parameter is induced along with the event types. Therefore, event *book*(*bookid*, *librarycard*) becomes a parameterized event type and the specification of

the *LibraryMonitor* diminishes as depicted in Fig2.3.4. Here both *bookid* and *librarycard* are explicit parameters but for event composition only *bookid*, an explicit key parameter is considered where as *librarycard* is an explicit non-key parameter i.e., event monitor is parameterized only on *bookid* and not on *librarycard*.

```
monitor LibraryMonitor{
primitive{
    event book<ptype : bookid>(int bookid, int librarycard).
    event arrive<ptype : bookid>(int bookid).
    event delay.
   }//end of primitive event declarations

task taskOne<context : chronicle>{
    event correct1<ptype : bookid>(int bookid, int librarycard) is
      N<expire:30 days>(book,delay,return).
    event incorrect1(int librarycard) is
      N(book, return, book+ 30days).
   }//end of taskOne
  }//end of monitor
```

Figure 2.3.4: Parameterized event monitor specification example

2.3.1 Benefits of Parameterized Event Composition

The benefits of Parameterized event composition are given below:

1) One can perform event composition based on a set of specific parameters that are of interest with respect to a specific application.
2) Correct event composition as illustrated in the example 2.3.1 can be provided.
3) In general, the non-parameterized event composition emulating parameterization needs more storage as all combinations of composite events are generated and then the process of filtering of composite events take place whereas in parameterized event composition only the contributing events having same parameters are contributed to form a composite event.
4) Parameterized event monitor specification is scalable and feasible in comparison to non-parameterized event composition as already explained in the previous section.
5) The size of an event monitor specification can be reduced (Mellin, 2004, p. 229) as already explained in the above section.

The concept of parameterization becomes significant provided that the systems performance does not deteriorate in terms of scalability, efficiency and predictability that is the results should be acceptable in terms of scalability and efficiency. The formal specification of parameterized event specification language is considered for future work since several issues such as homogeneity, orthogonality and symmetry must be supported as mentioned by Zimmer and Unland (1999). Also the event specification languages should not lack in systematic and comprehensive analysis of complex events.

3 Problem Definition: Investigation of Parameterized Event Composition

This section describes the purpose of parameterized event composition, the motivation behind, the steps that are followed for fulfilling the aim (objectives), the hypothesis and also about pre-requisites and assumptions.

3.1 Purpose

The aim of this work is to design a parameterized event composition that is efficient, scalable and resource predictable such that it can be widely used in various applications which can utilize the benefits of parameterized event composition mentioned in section 2.3 (p. 9).

3.2 Motivation

The concept of parameterization lacks in several event specification languages as already mentioned in section 2.3 (p.7) and in those having this feature did not address any of the factors such as efficiency, scalability and resource predictability. The impact of parameterized event composition may affect the system in terms of efficiency and scalability as condition for parameter equivalence of the contributing events is checked. Thus, analysis of parameterized event composition need to be made before we come to a conclusion that parameterized event composition is feasible in real-time applications having practical sizes of filtered event logs, event type sizes, etc. Secondly, the current available algorithms and data structures of non-parameterized event composition need to be refined before they can be utilized for parameterized event composition. Thus, there is an utmost need to design an efficient, scalable and resource predictable parameterized event composition.

3.3 Objectives

The following objectives need to be met in order to design an efficient and scalable event composition with the features such as correct event composition and a reduced size of event monitor specification and they are given below:

1. **Management of Filtered Event Log:** Monitor collects and stores large numbers of event occurrences in the filtered event log. It needs to analyze and signal these events to the subscribers to handle the tasks. To perform analysis in an efficient way, we need indexing technique. This is because the filtered event log is virtually divided into a set of queues where each queue refers to a set of events having same event type. So, to access a particular event from a particular queue we need an efficient main memory based indexing technique (e.g., trees, hash tables) that suits the functionality of event monitor. Secondly, partitioning of filtered event log may also be implemented to achieve efficiency and further details of partitioning are presented in section 4 (p.13).
2. **Evaluation of Parameterization in Event Composition:** Evaluate the event composition in terms of scalability and efficiency. Compare parameterized event composition with respect to non-parameterized event composition in terms of scalability and efficiency.

3.4 Hypothesis

The key hypotheses of this work are as follows:

1. **Feasibility Hypothesis:** It is feasible to construct parameterized event composition algorithms that enable monitoring of event expression in any event context of Solicitor event specification language defined by Mellin (2004, section 6.3).
2. **Scalability Hypothesis:** It is possible to design the aforementioned event composition algorithms to scale polynomial in time with respect to all significant input domains provided an efficient indexing technique is adopted for analyzing the filtered event log.

3.5 Pre-requisite and Assumptions

This section lists pre-requisite and a set of assumptions that are made in this research work and are described below.

Pre-requisite:

1. To implement an indexing technique, we need queues and parameters since indexing is done on parameters of event types where events containing parameters are mapped to the queues (a subset of filtered event log). So, both parameters and queues are required.

Assumptions:

1. The tasks performing event monitoring in a system know the bounds of resource requirements i.e., resource predictability is maintained since this work is based on DeeDS prototype (Andler et al., 1996) and Solicitor, an event specification language developed by Mellin (2004).
2. To resolve conflicts in event composition it is assumed that the event types are ordered and no two related events have equal time stamps.
3. All functionality and data collection are assumed to execute in tasks on the same resources as the tasks of the monitored object (an on-line monitor is considered).

The next section describes the architecture of parameterized event monitor, the need for indexing technique and partitioning scheme, the algorithm for parameterized event composition and its time complexity.

4 Algorithm and Data Structures for Composite Event Detection

Architecture of non-parameterized event composition is described before we present the architectural details of parameterized event composition. Consider an event expression 4.1, where $E_1, E_2, E_3, \ldots, E_n$ are primitive events and Ω is an event operator.

$$E=((((E_1 \, \Omega_{1,2} \, E_2) \, \Omega_{2,3} \, E_3) \ldots \Omega_{n-2,n-1} \, E_{n-1}) \, \Omega_{n-1,n} \, E_n) \quad \rightarrow (4.1)$$

This event expression can be represented in the form of a dataflow machine diagram as depicted in Fig. 4.1. A dataflow machine is a programmable computer where the hardware is optimized for fine-grain data-driven parallel computation (Arthur, 1986). The data-flow computer architectures have been suggested as a means to achieve increased computational throughput via the automatic detection and exploitation of potential parallelism. This process of evaluating the event expression using the dataflow machine diagram for detecting composite events exists in several languages such as Snoop (Chakravarthy et al., 1994), REACH (Buchmann et al., 1995), Solicitor (Mellin, 2004).

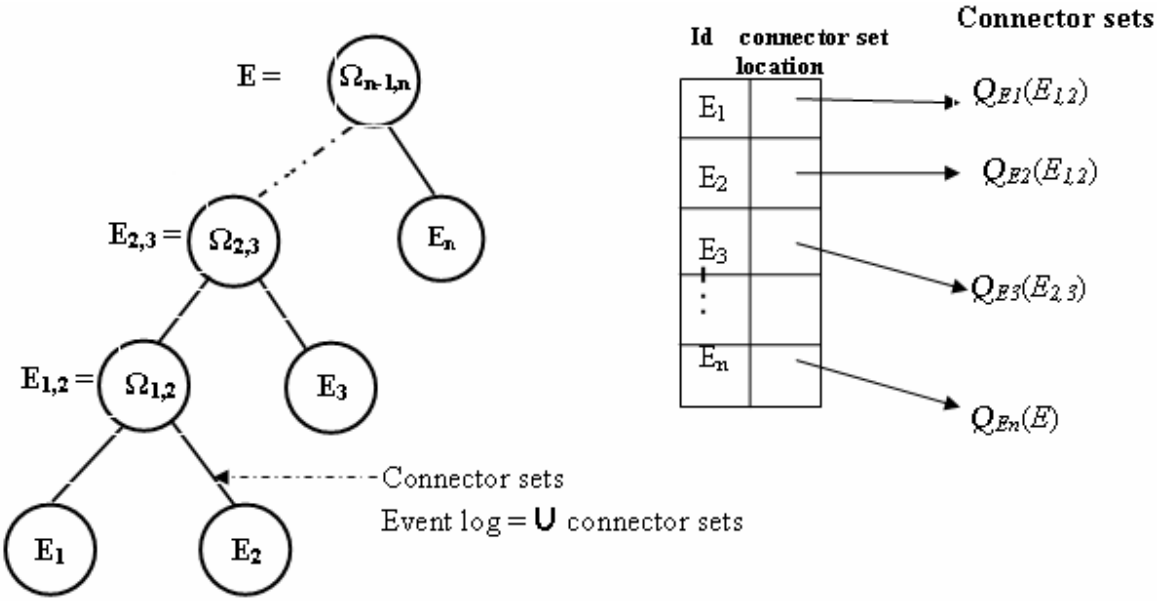

Fig 4.1: Dataflow Machine for Non-Parameterized Event Composition

The nodes in the dataflow machine represent either primitive event or composite event types. A node of event type *E* is triggered when an event of type *E* has occurred as an input to it and in turn this triggered node acts as an input to the parent node of *E*. In this way a composite event is detected when several events trigger the corresponding contributing events of a composite event. To perform this, each event occurrence in a real-time system need to be placed in the correct node position of a dataflow machine. In the case of event composition if several events of same event type occur then these events are placed in their corresponding connector sets as shown in Fig. 4.1. A *connector set* contains event occurrences that enable event composition of a contributed event type. Connector sets are only defined between an event type and the contributing event types of the event type's principal event operator (Mellin, 2004, in section 7.1). Let $Q_{E'}(E)$ be the connector set between the contributing event type *E'* and the event type *E*. The filtered event log is assumed to be realized as a union of connector sets i.e., filtered event log= $\cup$ Connector Sets. An efficient indexing technique is needed as we need to place the event occurrences in the correct connector sets and several lookup operations are required during event composition process for forming a composite event. The lookup table for this example is depicted in Fig.4.1 where event type forms the key and the location of the connector set holding several event occurrences forms the value. To achieve efficiency, event composition can be implemented using a) indexing and partitioning and b) indexing and non-partitioning. The Fig.4.2 depicts the partitioning of dataflow machine of Fig.4.1 into two tasks and thus resulting into two dataflow machines: one handling events related to task1 and the other handling events related to task2. A proxy node $\mathbb{E}_{1,2}$ and the corresponding connector set highlighted in the Fig.4.2 are required in order to locate the actual node $\mathbf{E_{1,2}}$ and connector set of a dataflow machine performing the task2. The Fig.4.2 contains two lookup tables: one for task1 and the other for task2. To find a particular connector set two lookup operations are required in this case i.e., one for finding the table (task) and the other for finding the actual connector set. This may cause further overhead as most of the processing time is consumed in determining the hash function or in traversing through the tree index structure for every lookup operation. On the other hand this process of partitioning may also improve the

efficiency of event composition as we are dealing with only a part of the filtered event log but the decision whether to adopt non-partitioning with indexing and partitioning with indexing is again application dependent. For small applications involving small sizes of event history, it is no good to partition the dataflow machine but in the case of large applications, it is good to partition the dataflow machine provided a good indexing technique is adopted for locating the connector sets and the lookup tables.

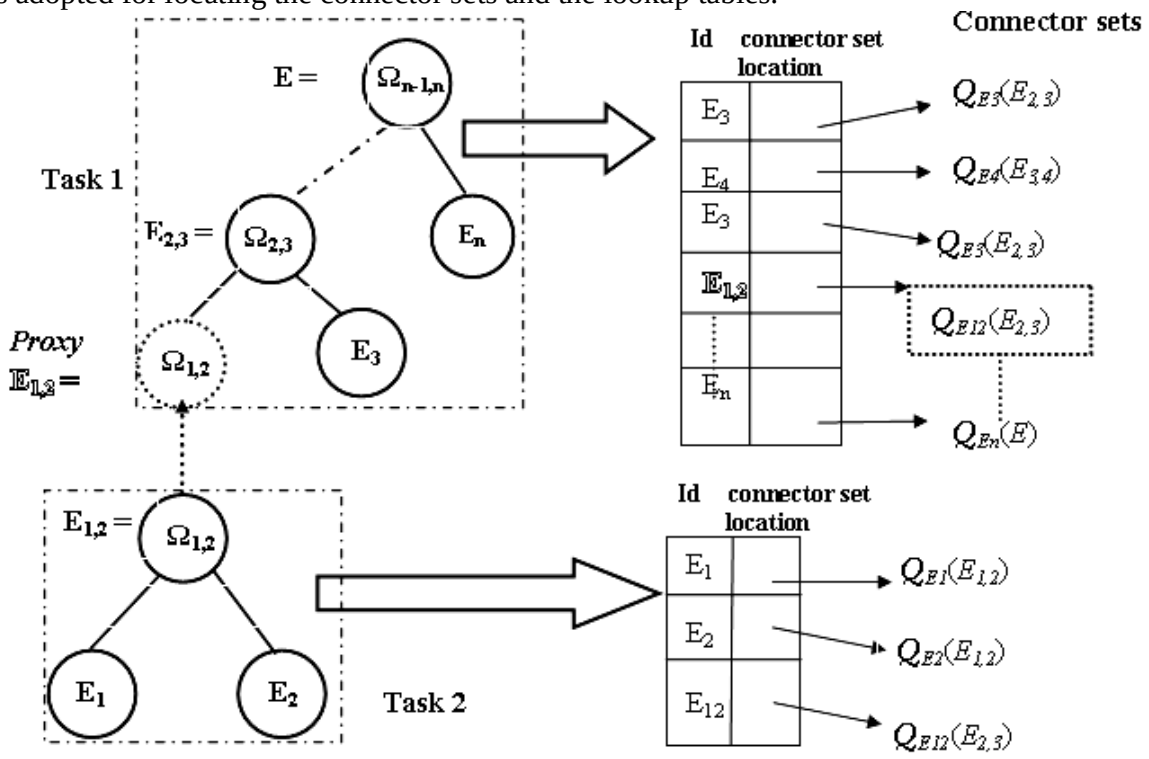

Fig 4.2: Partitioning of Dataflow Machine of Event Composition

Thus, the partitioning of dataflow machine and partitioning of filtered event log are correlated. If we partition the dataflow machine based on tasks then it restricts the possibility of partitioning a filtered event log efficiently. The advantage of this partitioning scheme is one can have distributed nodes and is also applicable to the systems having nodes with no shared memory. The efficiency of event composition may be improved because time taken to access an event occurrence is less in case of indexing implemented on a partitioned filtered event log.

4.1 Architecture of Parameterized Event Composition

The architecture of the parameterized event composition for event expression 4.1 is depicted in Fig 4.3., where event composition based on a parameter (for e.g., book identifier for the example 2.1.) is defined.

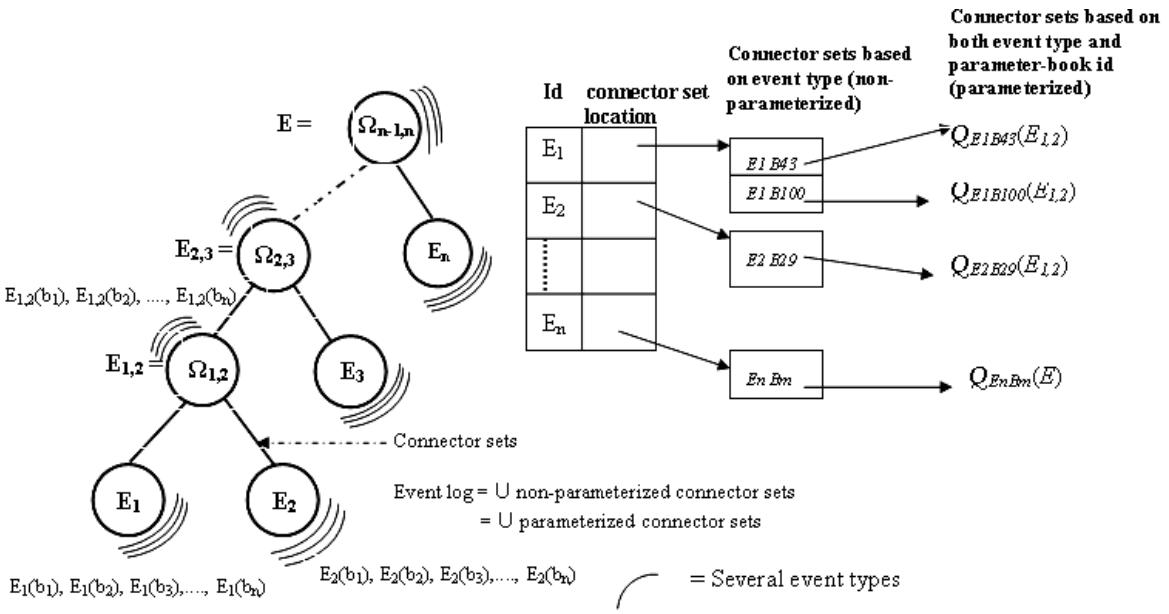

Fig. 4.3: Dataflow machine Parameterized on parameter: Book identifier

The major difference between dataflow machines constructed for parameterized event composition and non-parameterized event composition is in connector sets because each node represents several event types. Each connector set of non-parameterized event composition ($Q_{E'}(E)$)is further divided into a set of connector sets ($Q_{E'p}(E)$) based on parameter. Each connector set in parameterized event composition contains event occurrences with same parameter values (e.g., same book identity). Thus, $Q_{E'}(E) = \cup_{p \in P} Q_{E'p}(E)$, where P is a set containing different parameter values (e.g., book identities). The contributing events and the formed intermediary composite events need to be placed in the correct connector sets based on parameter value. To locate a connector set, suppose $Q_{E1B45}(E_{1,2})$, two lookup operations are required. One for locating the connector set that holds only E_1 event occurrences and the other lookup is for finding the actual connector set (i.e., $Q_{E1B45}(E_{1,2})$) among the resulted set of connector sets ($Q_{E1B45}(E_{1,2})$ and $Q_{E1B100}(E_{1,2})$) obtained from the first lookup. Thus, two maps are maintained, one map ($\boldsymbol{M}_{E'}(E)$) is defined between event type E' and connector set and the other map is defined between parameter k of a particular event type E' and the connector set ($\boldsymbol{M}_{E'}(E)[k]$).

4.1.1 Need for Indexing Techniques

To evaluate an event expression, several lookup operations need to be made for finding the correct connector set from which contributing events are used to form a composite event. For this, an efficient indexing technique is required. The efficiency and scalability of parameterized event composition is based on the indexing technique and also on the key on which indexing is done. The operations search, insert and delete for an event type should not end up with large time complexities resulting in more missing deadlines. In the next sub-section we discuss several main memory based indexing techniques that are suitable to the problem domain. We emphasize the concept of indexing because naïve solutions, such as using no index, degrade the performance of event monitor significantly as demonstrated in the BEAST benchmark (Geppert et al., 1998). We also discuss the possible keys on which indexing can be performed.

4.1.2 Available Indexing Techniques

The main memory based indexing techniques are considered because it is assumed that the filtered event log resides in main memory instead of disks. This is because accessing events from disks may lead to missing deadlines as the time taken to find and transmit a memory page is unpredictable. The management of disk accesses and disk storage results in inefficient usage of CPU cycles. With the rapid decrease in the cost of main memory it is feasible to place large event logs in main memory. Essentially there are many data structures available for consideration as index structures. One can classify index structures into two types: those that preserve some natural ordering in the data and those that randomize the data. The following order-preserving class of index structures is considered.

1. AVL Tree: The AVL Tree was designed as an internal memory data structure. It uses a binary tree search. Updates always affect a leaf node and may result in an unbalanced tree. The idea of balancing a search tree is given by Adelson-velskii and Landis (1962). Lehman and Carey (1986) addressed major disadvantage- poor storage utilization in AVL trees.
2. Red-black Tree: It is invented by Bayer (1972) and Guibas and Sedgewick (1978) studied their properties and introduced the red black convention. This tree is an elegant search tree scheme that guarantees O(log n) worst case running time in the case of insert, delete and search operations and also the computational cost for detecting an unbalanced tree is low (Cormen et al., 1993, p.272 and p.277).
3. Finger Tree: It performs better than balanced trees when the tree operations exhibit some locality of reference (Booth, 1992). In finger trees (Tarjan and Van Wyk, 1988) data is accessed through pointers to a fixed number of leaves rather than through the root. This makes the access time dependent on the distance to the leaf rather than on the number of items in the list. Concatenating and splitting operations are more complex.
4. Splay Tree: It is introduced by Sleator and Tarjan (1985). Splay operations involving rotations are performed within the tree every time an access is made with an assumption that the accessed item is likely to be accessed again soon. The disadvantages with this self-adjusting structures are:
    a) They require more local adjustments, especially during accesses(lookup operations)
    b) Individual operations within a sequence can be expensive which may be a drawback in real-time applications.
5. Randomized Splay Tree: The randomized splaying scheme (Albers and Karpinski, 2002) has the same asymptotic performance as the original deterministic scheme but achieves smaller constants in the O notation. The proposed scheme improves 25% over deterministic splaying provided the request sequences are generated by fixed probability distributions otherwise the improvements are negligible.

The following randomizing class of index structures is considered:

1. Chained Bucket Hashing: The chained bucket hashing (Knuth, 1973) is a static structure (fixed number of buckets) used both in memory and on disk. It is very fast because it is a static structure and never has to reorganize its data. The disadvantage of this hashing mechanism is addressed by Lehman and Carey (1986) and it behaves poor in a dynamic environment because the size of the hash table must be known or guessed before the hash table is filled.

2. Linear Hashing: A linear hash table (Litwin, 1980) grows linearly as it splits nodes in pre-defined linear order. This scheme can waste space with empty nodes (when a node corresponding to a hash entry has no data items).
3. Extendible Hashing: Fagin et al. (1979) introduced this technique and it employs a dynamic hash table that grows with the data. The problem with extendible hashing is that any node can cause the directory to split so the directory can grow very large if the hash function is not sufficiently random (Lehman and Carey, 1986).
4. Minimal Perfect Hashing: Minimal perfect hash functions are used for efficient memory storage and fast retrieval of items from a static set. A perfect hash function is an injection mapping from W→I where W contain a set of 'm' keywords and I contain a set of 'k' integers. The algorithm generates order preserving minimal perfect hash functions by generating random graphs (Czech et al., 1992). They state that it runs very fast in practice but Ho (1994) has implemented minimal perfect hashing and the observations of the experiment state that the hashing performance is slower than the open-addressing scheme.
5. Open-Addressing Hashing: In this technique, all elements are stored in the hash table and it resolves the problem of collisions. If a collision occurs, alternative cells are tried until an empty cell is found (Knuth, 1973).
6. Dynamic Perfect Hashing: Dietzfelbinger et al. (1988) introduced a simple perfect hashing scheme but it has poor practical performance (Craig, 1998). The randomizing hashing algorithms developed by Fredman (1984) does not support insertion or deletion but only lookup on a static data set. However, the algorithm is modified to guarantee constant time for insertion and deletion as well as a constant time for lookup.

Reader is requested to refer to the algorithms and data structures for further details.
Red-black trees are chosen due to that *(i)* they are one of the most efficient data structures (Cormen et al., 1993) as the worst case running time is O(log n)in the case of search, insert and delete operations and also the computational cost required for determining an unbalanced tree is low in comparison to the other approaches. A red black tree balances in 2 or 3 rotations when ever an element is inserted or deleted. Thus the computational cost for balancing a tree is also low and, therefore, *(ii)* they exist in many standard implementations of indexing such as C++ Standard Template Library (Meyers, 2001) and Linux kernel (Bovet and Cesati, 2003).
Similarly an open addressing hashing can also be considered as it is also one of the efficient indexing techniques (Cormen et al., 1993) and is also implemented in C++ map template library although it is not standardized (Meyers, 2001).

Indexing is performed on parameter on which the event composition takes place. The possible mappings are given below:

Option 1: map <parameter, connector set>
Option 2: map <parameter, event occurrence>

The first option maps parameter onto connector sets where we can access events by iterating through the elements of the connector sets. The second option directly points to the required event occurrence without the need for connector sets but the problem with this option is hashing cannot be implemented as hashing cannot handle ordered set of data and we

consider an ordered set of events for event composition. An ordered set of events based on increasing orders of time stamps are considered for the event composition. The former option can be implemented both in the case of hashing and red-black tree. The height of the tree or the length of the hash table will be less in comparison to the second option since the second option inserts each event occurrence occurring in the system. The event logs of real applications are large and this results in an over head in maintaining a hash table or a tree structure.

4.2 Algorithm for Parameterized Event Composition in Binary Operators

The algorithm performing parameterized event composition is depicted in Fig.4.4. This algorithm is based on non-parameterized event composition algorithm addressed by Mellin (2004, section 9.1 and 9.2). Let ξ_m be the set of monitored event types. Each event type from a set of monitored event types is iterated over once for each rule associated with the principal operator of the event type. In general, $M_{E'}(E)$ is a map containing key value pairs equal to the parameter{key}of an event type *E'* and the connector set $Q_{E'}(E)$ {value}. Here connector set is defined between the contributing events of type *E'* and the event expression *E* where the parameter values of the contributing events *E'* are equal. i.e., in one sentence connector sets hold contributing events having unique event type and unique parameter value. Get all the keys of the maps $M_{E1}(E)$ and $M_{E2}(E)$ into key sets $K_1$and K_2 respectively. This is performed in order to find the common keys between the left input map ($M_{E1}(E)$) and the right input map($M_{E2}(E)$) such that we can get the event occurrences having same parameter value from the left and the right connector sets. Here left and right can be visualized from the Fig. 4.3 where all the connector sets that are left to the event type *E* of any event expression form left connector sets and is same with the case of right connector sets. Store the common keys into a key set K by performing intersection of these two key sets K_1 and K_2(i.e., $K = K_1 \cap K_2$). Now, for each key '*k*' in this common key set find the corresponding connector sets Q_1 (left connector set) and Q_2 (right connector set) having E_1 and E_2 event occurrences with *k* as the parameter. A composite event *E* is formed provided there exists event occurrences γ_1 and γ_2 from their respective connector sets Q_1 and Q_2 satisfying the rules of the context predicate $\mathbf{R}_c$. The context predicate $\mathbf{R}_c$ ensures whether the candidate event occurrences can be used or invalidated according to a specific event context (Mellin, 2004, p. 110, definition 6.12). The formed composite event has an interval with the interval start equal to the start time interval of γ_1 (initiator) and interval end equal to the end time interval of γ_2 (terminator). The formed composite event *E* is in turn added to the connector set having event occurrences with parameter *k* and event type *E*. To add this composite event to a specific connector set a lookup operation on map $M_E(E_{parent})$ is needed to locate the connector set Q_E . This entire process changes the state of the monitor. The function all_operands(*E'*) returns all the immediate contributing event types of event type *E'*. The U_α refers to the set of initiator occurrences used to contribute to a composite event occurrence whereas U_ω refers to the set of terminator occurrences. The reader is requested to refer to the formal definitions defined by Mellin (2004, p.342).

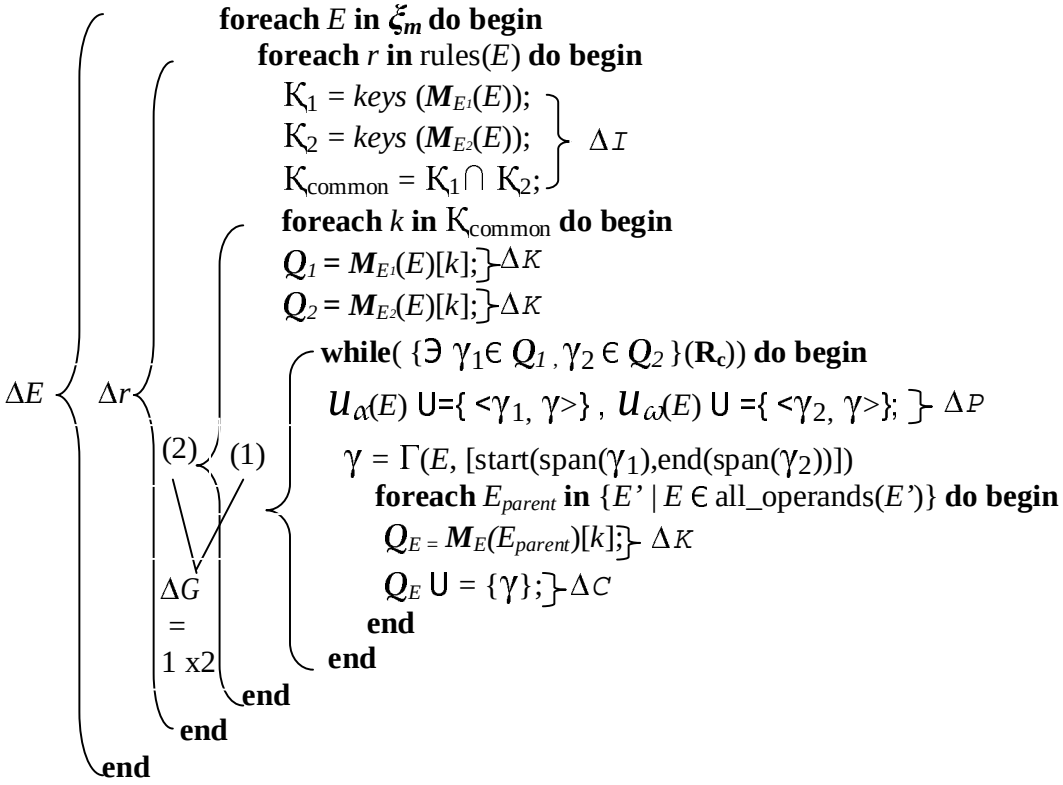

Figure 4.4: Algorithm performing parameterized event composition only

4.3 Algorithmic Time Complexity of Event Monitoring

According to Cormen et al. (1993), the algorithmic time complexity is defined to be the asymptotic efficiency of algorithms. In this section, we investigate the asymptotic efficiency of algorithms implementing event composition in different event contexts. This study of algorithmic time complexity emphasizes parameterized event composition since the time it takes to generate a number of composite event occurrences is significant. This is because, in parameterized event composition, in addition to the formation of composite event and pruning of contributing event occurrences, each composite event occurrence requires selection of contributing event occurrences having same parameter values. A model of parameterized event composition is presented in section 4.2, Fig 4.4. This model is based on the non- parameterized event composition model developed by Mellin (2004, section 9.1). So, this model has all the assumptions that are made in the case of non- parameterized event composition. Moreover, the analysis for determining the scalability of parameterized event composition follow similar set of steps defined by Mellin (2004, p. 197). The steps for determining the scalability are given below:

Step 1: Identify the sources of complexity and define the computational cost functions in terms of the complexity sources.

Step 2: Define the computational cost functions in terms of recurrence relations that define the number of computational steps.

Step 3: Solve the recurrence relation to result a closed function that represents the computational cost function of recurrence relation.

Step 4: Derive time complexity classes from these computational cost functions with respect to different significant situations and significant design choices.

The significant situation is whether there is a single primitive terminator occurrence (implicit invocation) or multiple primitive terminator occurrences (explicit invocation) for evaluation. The design choice is whether to employ safe or unsafe pointers to manage event occurrence parameters. Safe pointers avoid dangling pointers. They do not make a deep-copy of the complete event occurrence instead they pass only the reference of the event occurrence until the event occurrences are to be signaled across an address boundary. Since, the scalability of event monitoring is poor in the case of unsafe pointers only the case of safe pointers is employed in this thesis (Mellin, 2004, p.273).

Consider the parameterized event composition algorithm mentioned in Fig.4.4. The segment ΔE is iterated once for each event type. The segment Δr is iterated over once for each rule associated with the principal operator of the event type. Since only composition is considered, there are a constant number of iterations over rules per event type, because there are a constant number of eligible generation rules per operator. Although there are two loops in this algorithm (1 and 2 as shown in the Fig.4.4), the combination of these two loops contribute to the segment ΔG of Mellin (2004, p.199). This is because the filtered event log in non-parameterized event composition is divided into a set of connector sets where as in parameterized event composition, these connector sets are further divided into a cluster of connector sets based on the parameter on which indexing is performed (i.e., Filtered event log = $\bigcup Q_{E'}(E)$ {connector sets in non-parameterized event composition}= $\bigcup_{p \in P} Q_{E',p}(E)$ {connector sets in parameterized event composition}. The segment (2) iterates $|K_{common}|$ times (the number of keys that are common to both the contributing events of the connector sets) multiplied by the number of contributing pair of eligible initiators and terminator event occurrences obtained from the clustered connector sets. This in turn results to the same number of iterations made in ΔG segment of non-parameterized event composition. Additional segments ΔI and ΔK are considered for obtaining the common key set with the parameter values, finding keys that are common(ΔI) and lookup operation (ΔK) needed to find the correct connector set that participates in event composition respectively. The cost for obtaining the common key set is assumed to be kf(k). The cost of locating the connector set is assumed to be a function of k i.e., f(k), where k is an integer variable denoting the number of keys, a map $\boldsymbol{M}_{E'}(E)$ has.

Definition 4.1

Cost for locating a Connector Set:

$$f(k) = \begin{cases} O(\log(k)) & \text{iff red-black tree index structure used for mapping} \\ O(1) & \text{iff hash index structure used for mapping} \end{cases}$$

The segments ΔP and ΔC are insignificant as they are iterated only for once for each generated composite event (Mellin, 2004, p. 200).

The selection of event occurrences for matching is based on the context predicate and only these predicates need to be studied for algorithmic time complexity.

4.3.1 Sources of Complexity

The sources of complexity in event composition are defined in table 4.1. The variables that are in italic text denote the size of some input such as *e* denoting the variable for number of operators and upper case variable within '|' denotes a size of something (e.g., $|E|$ denotes the size of an event type.

Name	Variable	Constant	Segment	Comment
Size of filtered event log	g	$\|G\|$	ΔG	g is the number of primitive event occurrences in the filtered event log of which some are initiators (i) and some are terminators (t).
Event type size	e	$\|E\|$	ΔE	The number of operators in the event expression.
Number of monitored event types	m	$\|\xi_m\|$	ΔE	The number of event types that are monitored.
Size of the hash table/ tree data structure	k	$\|K\|$	ΔL	Number of keys in a map $\boldsymbol{M}_{E'}(E)$.
Size of Parameters	p	$\|P\|$	ΔP	The physical size of parameters per primitive event type.

Table 4.1: Variables for Time Complexity Investigation

These variables are considered in the investigation of time complexity because it is interesting to know how event composition depends on event history, event expression, number of keys/unique parameter values and the size of parameters because more steps may have to be taken to generate composite event occurrences if there are more event occurrences in the filtered event log (*g*). The more contributing event types there are, the more steps may have to be taken to complete event composition processing (*e*). The third source of complexity is the number of monitored event types (*m*). The more event types that are monitored, the more iterations may have to be performed by the evaluation algorithm. Here *m* represents all event types in the system. The fourth source of complexity is *k* and event composition may take more steps if the number of keys (i.e., the number of unique parameter values) is large. More the number of keys, more the number of lookup operations are needed. If the domain size of the parameters is large then the number of monitored event type's *m* is a subset of this domain. The last source of complexity is the physical size of parameters for each primitive event type (*p*). The more data that is carried the longer the time it takes to process data.

4.3.2 Algorithmic Time Complexity for Event Contexts

In this section we present the time complexity of event composition in event contexts- chronicle, recent and continuous. The step 1 mentioned in section 4.3 (p. 18) is performed in section 4.3.1 and steps 2, 3 and 4 are discussed in this subsection. The total cost function for event monitoring is presented in Def. 4.2 based on the monitoring cost function defined by Mellin (2004, definition 9.4).

Definition 4.2
Monitoring cost function:

$$\text{Monitor}(g, e, m, k, p) = \text{lookup}(g, e, m, k, p) + \text{composition}(g, e, m, k, p) + \text{prepare\_delivery}(g, e, m, k, p)$$

Definition 4.3
Lookup cost function:

$$\text{lookup}(g, e, m, k, p) = \begin{cases} O(g\, f(m)f(k)) & \text{iff multiple terminators} \\ O(f(m)f(k)) & \text{iff single terminator} \end{cases}$$

The cost function lookup is the computation cost of finding the connector set for each primitive event occurrence that need to be added to. The *m* connector sets in non-parameterized event composition are further divided into *mk* connector sets having unique event type and parameter value. So, the computation cost of finding a connector set for a signaled primitive event occurrence of event type '*E*' is f(*m*)f(*k*) because two lookup operations are made: one for finding connector set that holds event type '*E*' and a second lookup is for finding a connector set holding event occurrences of event type '*E*' and with a parameter *k*. This cost f(*m*)f(*k*) is multiplied by *g* if there are multiple event occurrences that need to be evaluated.

Definition 4.4
Composition cost function:

$$\text{composition}(g, e, m, k, p) = s_e$$
$$\text{where } s_e = \text{get\_common\_keyset}(k) + k(\text{find\_contributing\_connectorsets}(k) + \text{context}(i_{e-1}, t_{e-1})\text{generate\_store}(e, k)) + s_{e-1}$$

s_e defines the number of computational steps that have to be performed for evaluating an expression of *e* event operators. The major computational steps that involved in evaluation with respect to the Fig. 4.4 are:

1. Obtaining the common key set (segment ΔI). The cost function for this is *k*f(*k*).
2. For each common key *k*, the loop iterates through the segments ΔK + (1)times the segments (ΔP+ΔK+ΔC). So, the cost function is *k*(find_contributing_connectorsets + context(i_{e-1},t_{e-1})generate_store(*e*))

The variable i_{e-1} can be substituted with *i* because when the size of the filtered event log approaches ∞, then the total number of initiators as well as number of initiators per contributing event type approaches ∞. Thus, the simplified recurrence relation is given below:

$$s_e = \text{get\_common\_keyset}(k) + k(\text{find\_contributing\_connectorsets}(k) + \text{context}(i, t_{e-1})\text{generate\_store}(e, k)) + s_{e-1} \quad \rightarrow \quad (4.2)$$

Definition 4.5
Get Common Key Set cost function:

$$\text{get\_common\_keyset}(k) = kf(k)$$

This cost refers to the cost caused by the segment ΔI (Fig.4.4) where the common keys from both the contributing event types are determined.

Definition 4.6
Find contributing connector sets cost function:
find_contributing_connectorsets(k) = f(k)
For each common key, connector sets of the contributing events need to be searched and this search may cost $O(\log k)$ in the case of a red-black tree and $O(1)$ in the case of a hash data structure.
Definition 4.7
Generate_store cost function:

generate_store(n, k)= 1+ f(k) iff safe pointers are used

An additional cost of f(k) is added to the generate cost function (Mellin, 2004, Def.9.7) of non-parameterized event composition. This is because a lookup operation is required to locate a connector set that holds the composite event occurrences of event type equal to the event type of the generated composite event and parameter value equal to 'k'.
The cost functions for context(i, t_{e-1}) and prepare_delivery(g, e, m, k, p) are same as the definitions mentioned in Mellin(2004).
Definition 4.8
Context cost function:

$$\text{context}(i, t_{e-1}) = \begin{cases} t_0 & \text{iff } e = 1 \\ \text{composed}(i, t_{e-1}) & \text{iff } e > 1 \end{cases}$$

[Def. 9.8 of Mellin(2004, p. 208)]

Definition 4.9
Number of generated event occurrences function: [Def. 9.9 of Mellin(2004, p. 208)]

$$\text{composed}(i, t_{e-1}) = \begin{cases} t_{e-1} & \text{for chronicle event context} \\ 1 & \text{for recent event context} \\ i & \text{for continuous event context} \\ it_{e-1} & \text{for general event context} \end{cases}$$

Definition 4.10

$$\text{prepare\_delivery}(g, e, m, k, p) = \begin{cases} O(ep) & \text{iff single terminator} \\ O(gep) & \text{iff multiple terminators} \end{cases}$$

The general simplified recurrence relation in Eq. 4.2 on p.21 can be rewritten as in Eq. 4.3 by replacing the functions: get_common_keyset(k), find_contributing_connectorsets(k) and generate_store(e, k) with their definitions.

$$s_{\hat{e}} - s_{\hat{e}-1} = k\text{f}(k) + k\big(\text{ f}(k) + \text{context}(i, t_{e-1})\ (1+ \text{f}(k))\big) \quad \text{iff safe pointers} \quad \rightarrow \quad (4.3)$$

In chronicle, recent and continuous event contexts, the closed form functions can be derived from the non-homogeneous difference equations. The obtained closed form functions define composition cost functions that in turn defines monitoring cost function. The derivation for the composition cost function for chronicle context is given in Appendix A. The table 4.2 depicts the time complexity of event monitoring for 3 event contexts: chronicle, recent and continuous.

Event Context	Single terminator-Safe pointer	Multiple terminator - Safe pointer
Chronicle	$O(f(k)(f(m)+epk))$	$O(gf(k)(f(m) + epk))$
Recent	$O(f(k)(f(m)+epk))$	$O(gf(k)(f(m) + epk))$
Continuous	$O(f(k)(f(m)+gepk))$	$O(gf(k)(f(m) + epk))$

Table 4.2: Time Complexity of Parameterized Event Monitoring in contexts: chronicle, recent and continuous

From this table we can observe that the time complexity of recent context is same as for chronicle event context. The reason is that any optimization for recent event context does not give any effect on the complexity class (Mellin, 2004, section 9.4.4). Secondly, the time complexity of continuous event context is same as chronicle context in the case of multiple terminators but varies in the case of single terminator because the time complexity of event composition in continuous event context is always dependent on the size of the filtered event log (g).

4.3.3 Relevance of study of Algorithmic Time Complexity of Event Monitoring

This study of algorithmic time complexity is important, since it provides guidelines of how event contexts affect the processing of event composition. Out of the sources of complexity, the size of the filtered event log (g) is likely to be the most significant since systems of real applications signal few thousands of event occurrences. The response time for both recent and chronicle event contexts are $O(1)$ with respect to g for processing a single terminator occurrence.

In contrast e is not considered to be significant since most event expressions contain at most 2-5 operators (Berndtsson et al., 1999). The p is insignificant in the case of safe pointers because instead of passing complete copies of objects, only a reference to the object is passed as an argument in the procedure calls. This design concept is used in the experiments of section 5.2 conducted for validating the theoretical model.

The k is considered to be significant if the number of unique parameter values or the size of keys is large. This is because more lookup operations are needed if the size of keys is large. More lookup operations result in more time in evaluating the event expression. The variable m becomes insignificant when the domain size of the parameter is large. So, we investigate on these significant parameters and how they affect the algorithmic time complexity of event monitor in the case of safe pointers and explicit invocation (multiple terminators). In *explicit invocation*, the evaluation of an event expression is explicitly invoked after an arbitrary number of primitive event occurrences have been signaled (Mellin, 2004, p. 153).

To summarize, event contexts: chronicle, recent, and continuous scales, at worst, polynomial with respect to g, e, p and k for safe pointers and for processing of multiple terminator occurrences.

5 Experimental Results

In general an experimental model is epitomized by the scientific method: observe the world, propose a model or a theory of behavior, measure and analyze, validate hypotheses of the model or theory (or invalidate them), and repeat the procedure evolving our knowledge base (Basili, 1996). So, it is desirable to validate the accuracy of the theoretical model of event composition (in section 4.2) with respect to a real implementation, since event composition is a complex task performing different basic operations. Experiments are chosen to validate this accuracy, since it is difficult to analyze the code.

5.1 Implementation Details

The parameterized event composition is implemented on the existing event monitor implemented in C++ for the Solicitor event specification language. This event monitor is a separate software component and it can be used without the database management system. Task management, memory management, and message passing used by the event monitor is provided by the DeeDS operating systems interface that is an adapter ported to OSE Delta, Solaris, and Linux. In this experiment, the underlying operating system is the OSE Delta 3.2 kernel running in emulation mode on a host target with full error detection and system debugging turned on. The host target is IBM (300 MHz) running Linux 3.0. To compile the C and C++ code, the GNU compiler version 2.8.1is used. All the experiments were performed with debugging information compiled into the monitor to enable location of faults.

Few assumptions are made during the actual implementation of the event composition process and they are:

1. The information regarding size, data type and the relative position of a parameter among several parameters must be known because a composite event results only if the contributing events have same parameters.
2. It is assumed that parameters are specified in the same order and are of same data type in all the primitive event types declared in an event expression invoking the method defining event composition mechanism.

This approach does not handle parameters with a continuous value domain i.e. event monitor handles only discrete parameters with a limited range. The experiments are conducted only for chronicle context and for a binary operator (sequence) as it is assumed that the implementation of chronicle context is complex and also implemented in most of the real applications (Buchmann et al., 1995). One can also study the effect of event composition with respect to event history as it is fairly large in the case of chronicle context. Almost all the events of the filtered event log participate in the event composition process where as in the recent context only the recent initiators are considered and thus reduces the size of the filtered event log effectively. Secondly, binary operator (sequence) is considered as it is one of the most complex event operators and also if event composition scales better then it also works fine with unary operators since binary operators are more complex than unary operators. Each coordinate in the graph is obtained by dividing the average of all the response times obtained for 10 experiments by the number of event occurrences in the event history (i.e., event history size). The average response time per event occurrence is considered and the repetition of each experiment is about10 times. This is done in order to avoid results based on outliers and problems related to clock granularity.

5.2 Validation of Algorithmic Time Complexity

This section demonstrates that the average response times of parameterized event composition scales in the same way as the average response times of non-parameterized event composition with respect to the event history and further addresses the dependency between the average response time and the variables history size and key size.

5.2.1 Average Response Times of Non-Parameterized and Parameterized Event Monitoring

The graph in the Fig. 5.1 demonstrate that the average response times of parameterized event monitoring and non-parameterized event monitoring scale in the same way. This was measured for the following configurations:

- Event context: Chronicle
- Event History Size (| *G* |): 1000,2000,3000, …,100000
- Event Type Size | *E* |: 1
- Key Size | *K* |:1024
- Parameter Size | *P* |: 1
- Pointer Type: Safe
- Requested repetition: 10

The graph depicts that the average response times of parameterized event monitoring is more than non-parameterized event monitoring. This is an expected result. Also as expected (Fig B.1 in Appendix B) for small sizes of event history, the average response times per event occurrence do not follow the theory of time complexity of event composition (i.e., the average response time per event occurrence remains constant irrespective of the size of event history, $O(g\,f(k)(f(m) + epk))$). There might be one reason behind this drop in response time as in the initial stages, to process small history sizes, the initialization phase takes more time than the time taken for analyzing events (event composition) and as soon as the event history size grows to around 15000 and more, then there is not much to do with the initialization phase and the major part of the response time belong to the analysis part of event composition. For event history sizes greater than 15000 and less than 60000 (i.e., 15000<|G|<60000), it behaves as expected (i.e., average response times per event occurrence remain approximately constant with the increase of event history size). The total response time can be written as:

Total Response time= $kn + l$ where '*k*' and '*l*' are constants and
'n' is the number of events in the filtered event log.

For small sizes of 'n', *l* is significant (the initialization phase) and for large sizes of 'n', *k* (analysis part of event composition) is significant because average response time is:

Average Response time = $k + l/n$

For larger event history sizes (i.e., greater than 60000, |G|>60000), it seems to scale polynomial both in the case of non-parameterized and parameterized event monitoring but preserving the fact that the average response times of parameterized event monitoring are larger than non-parameterized event monitoring. The reason behind this increase in the response times might be due to paging in the case of main memory. To validate the theoretical model for these sizes of event history, the experiments need to be re-run with more memory.

The quality of data can be viewed from the Fig.5.2, where the average response times of parameterized and non-parameterized event monitoring are more or less the same in the

range of history size greater than 15000 and less than 60000 (i.e., 15000<|G|<60000). For large history sizes (|G|>60000), the experiments need to be re-conducted with more memory in order to avoid memory paging and to maintain the scalability feature. If non-parameterized event monitoring is acceptable then parameterized event monitoring may also be acceptable as the overhead is a fraction more in comparison to the non-parameterized event monitoring. However acceptance depends upon the application and the requirement (i.e., application dependent).

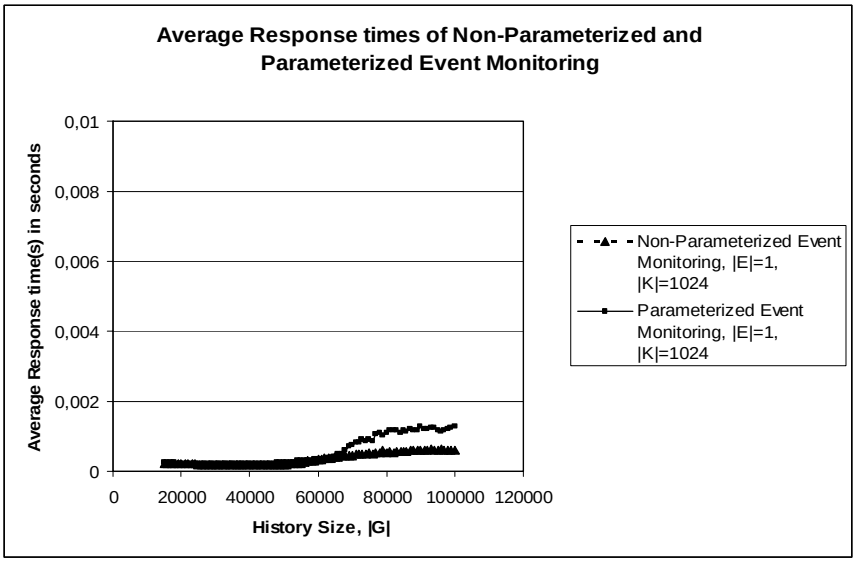

Fig. 5.1: Average response times of parameterized and non-parameterized event monitoring

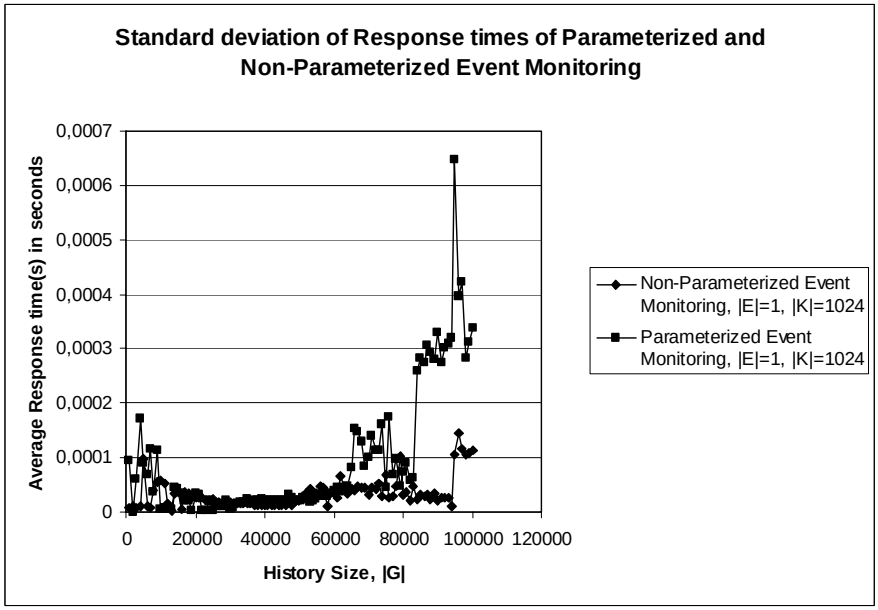

Fig. 5.2: Standard deviation of response times of parameterized and non-parameterized event monitoring

5.2.2 Dependency on Key Size

The effect of scaling the key size for a range of 100 to 5000 different parameter values was measured for the following configurations:

- Event context: Chronicle
- Event History Size (| *G* |): 5000
- Event Type Size | *E* |: 1,5,10
- Key Size | *K* |:100,200,300,…,5000
- Parameter Size | *P* |: 1
- Pointer Type: Safe
- Requested repetition: 10

The graph in the Fig 5.3 demonstrates that the experiment corroborate the theory that the event composition scales polynomial in worst case with respect to the key size *k*. The average response times per event occurrences remains same for both the event expressions |E|=5 and |E|=10 as depicted in the Fig 5.3. In theory, with the increase in the size of event expression, the average response times per event occurrence should also increase but in this case the average response times remained same for both |E|=5 and |E|=10. This is because the chosen range of |E| is less and theory may hold for large event expressions (for e.g., |E|=30). An unexpected result has also occurred even in this case i.e., as shown in the Fig. B.2, the average response times for |E| =1 is larger than the |E|=5 and 10. Further investigation on results is required as in general, the average response times should be small for small event expressions in comparison to large event expressions. One of the reasons behind this unexpected result is for small event expressions, the initialization phase of event monitoring process may consume more computational time and as the event expression grows (e.g., in this case |E|=5 as shown in the Fig B.2), the initialization phase becomes insignificant, resulting in low average response times.

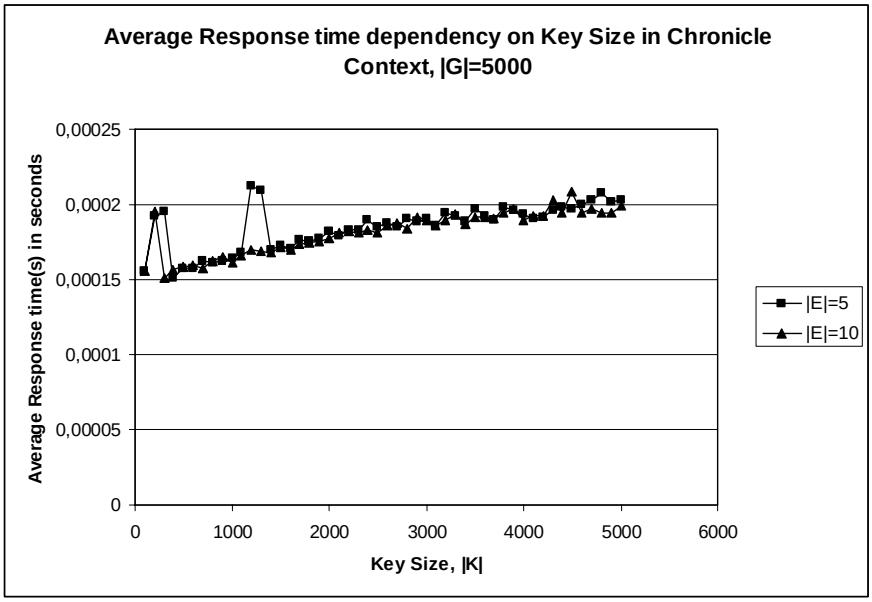

Fig 5.3: Response time dependency on Key Size in chronicle context

6 Comparison between Parameterized Event Monitoring and Non-Parameterized Event Monitoring

In this section we compare parameterized event monitoring with respect to non-parameterized event monitoring in terms of scalability and efficiency. The algorithmic time complexity of non-parameterized event monitoring in three event contexts: chronicle, recent and continuous is depicted in the Table 6.1.

Event Context	**Single terminator-Safe pointer**	**Multiple terminator-Safe pointer**
Chronicle	$O(\mathrm{f}(m) + ep)$	$O(g(\mathrm{f}(m)+ep))$
Recent	$O(\mathrm{f}(m)+ep)$	$O(g(\mathrm{f}(m)+ep))$
Continuous	$O(\mathrm{f}(m)+gep)$	$O(g(\mathrm{f}(m)+ep))$

A **B**

Table 6.1: Time Complexity of Non-parameterized Event Monitoring in contexts: chronicle, recent and continuous [based on Mellin (2004, section 9.4)]

When we compare the table 4.1 (time complexity of parameterized event monitoring) with the table 6.1(time complexity of non-parameterized event monitoring), we can clearly observe that there are two components (**A** and **B**) that are in common to each cell of the tables 6.1 and 4.1. So, the algorithmic time complexity of parameterized event monitoring in terms of non-parameterized event monitoring is of the form $\mathrm{f}(k)(\mathbf{A} + \mathbf{B}k)$. Thus, theoretically this will be the overhead in the case of parameterized event monitoring. The presence of overhead can also be viewed in the case of experiments as shown in the Fig. 5.1. So, parameterized event monitoring is less efficient in comparison to non-parameterized event monitoring but the advantages of parameterized event monitoring mentioned in section 2.3 p. 9 come at this cost. The scalability of parameterized event monitoring is same as the non-parameterized event monitoring as mentioned in section 5.2.1.

7 Discussion

The acceptance of the parameterized event monitoring depends on the application. If non-parameterized event monitoring is acceptable then parameterized event monitoring may also be acceptable since the proposed model exhibits same trend in terms of scalability and in the case of overhead, it is a fraction and it might be acceptable in real-time applications. Secondly, the proposed model is general and can be used by other monitor specifications provided they also have a similar implementation of connector sets (queues) and architecture of a dataflow machine that was discussed in the section 4.1. The proposed algorithm for parameterized event composition (Fig.4.4, p.18) deals with binary operators. It is possible to extend the algorithm to unary operators as the working mechanism of binary operator is more complex than unary operator and also the algorithm is general and scales better in the case of binary operators. The theoretical model and the experimental results validate the hypotheses (scalability and feasibility hypotheses) that were made but there was no much discussion about resource predictability in the previous sections. This is because resource predictability is maintained by the Solicitor prototype and is managed by dealing with the

expiration semantics of event occurrences. An efficient and scalable design of parameterized event composition not only depends on the partitioning and indexing mechanisms but also depends on the specification i.e. how efficiently one can specify a list of parameters and if specified, the resulted specification language must be validated such that there are no inconsistencies and redundancies in the event composition results. The specification language should also support the essential features mentioned by Zimmer and Unland (1999).

8 Related work

The concept of parameterization lacks in several event specification languages. Parameterized event types are dealt either explicitly or implicitly in five languages: SAMOS (Gatziu and Dittrich, 1992; Gatziu et al., 1994), TCCS (Baekgaard and Godskesen, 1997), REACH (Buchmann et al., 1995), Event Pattern Language (Motakis and Zaniolo, 1997) and Solicitor (Mellin, 2004). SAMOS an active database prototype based on Petri-nets supports parameterized event types having parameters as transaction identities and user identities. The specification and implementation of parameterized event types exists in SAMOS but in a very restricted form where parameterized event types are considered based only on same user and transaction identifiers. Baekgaard and Godskesen (1997) present a specification language that can specify real-time triggering conditions and can handle complex event patterns. They address parameterized event types in their specification language called TCCS but from the implementation point of view there does not exists any implementation of parameterized event types. Buchmann et al. (1995) presents REACH, a layered architecture for OODBMS. It implicitly mentions about parameterized event composition as it handles event composition of event occurrences occurring from different transactions i.e., the contributing event types involving in event composition have different transaction identifiers. Event pattern language has a similar concept of restricting the number of participating events in event composition by their semantic rules based on $DataLog_{1S}$. The event contexts are not parameterized and however, none of these works performed any study on algorithmic time complexity of event monitoring process in different event contexts. Only in Solicitor (Mellin, 2004), an extensive study on the affect of algorithmic time complexity with respect to different variables is performed. Secondly, the need for parameterization is identified by Mellin (2004) but there is no implementation of this concept. None of these specification languages have explicitly mentioned parameterized event composition and its affect on the event monitor if deployed. Also, factors such as efficiency, scalability and resource predictability of monitoring of events are not specified in any of these languages. This thesis is based on Solicitor work and deals with the implementation of parameterized event monitoring and also an extensive study on algorithmic time complexity is made.

9 Conclusion

The experiments suggest that it is feasible to perform parameterized event composition in real-time applications having practical sizes of filtered event logs. Secondly, it is feasible to construct parameterized event composition algorithms that enable monitoring of event expression in any event context of Solicitor event specification language (feasibility hypothesis). Currently, parameterized event composition is implemented for chronicle event context and event expression having sequence operators. This works fine as the results from

the experiments validate the theoretical model that is presented in this research work. So, this can be extended to any event context of any event expression in the case of Solicitor event specification language (Mellin, 2004, section 6.3).

9.1 Summary

To summarize, a feasible and a scalable parameterized event composition is presented in this research work. We present a theoretical model for parameterized event composition and calculated the algorithmic time complexity in order to validate the hypotheses that were made. The experimental results also validate the theoretical model and suggest that it is feasible to integrate parameterized event composition in event monitoring.

9.2 Contribution

A theoretical model representing parameterized event composition is presented. A study on significant sources of complexity and the algorithmic time complexity of parameterized event composition is presented. Issues on scalability and efficiency of the parameterized event monitoring are discussed.

9.3 Future work

The suggestions for future work are presented here:

1. Further investigation on parameterized event composition is required as the exact reason behind the drop in the curve of response times of Fig. B.1 need to be found. Secondly, the answer for the question: Is this overhead acceptable? need to be answered.
2. It would be interesting to specify a formal specification of parameterized event specification language which takes into consideration of the dimensions presented in the meta-model of Zimmer and Unland (1999).
3. Investigation on several indexing techniques is required as efficiency of the parameterized event composition depends upon the type of index structures that are implemented for the purpose of locating the connector sets.
4. In this thesis, partitioning of filtered event log based on tasks is discussed but not yet implemented. So, it would be interesting to implement partitioning techniques and investigate how the event composition process is affected.
5. In this thesis, the experiments are performed in order to validate the assumptions made for the proposed theoretical model of parameterized event composition. Experiments are implemented on a real monitor with a combination of simulated input of events. It would be better to investigate the efficiency of the parameterized event composition process in a real application. So, a case study on parameterized event monitoring can be performed.
6. It would be better to state the real-time applications for which this parameterized event monitoring is suitable.

References

Adel'son-ve'lskii, G.M. and Landis, E.M. 1962. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3, pp. 1259-1263.

Albers, S. and Karpinski, M. 2002. Randomized splay trees: Theoretical and experimental results. *Information Processing Letters*, 81, pp.213-221.

Arthur, H.V. 1986. Dataflow machine architecture, *ACM Computing Surveys (CSUR)*, 18(4), December, pp. 365-396.

Andler, S., Hansson, J., Eriksson, J., Mellin, J., Berndtsson, M., and Eftring, B. 1996. DeeDS towards a distributed active and real-time database system. Special Issue on Real Time Data Base Systems, SIGMOD Record, 25(1), March.

Bayer, R. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*. 1, pp. 290-306.

Basili, V.R. 1996. The Role of Experimentation in Software Engineering: Past, Current, and Future, *In Proceedings of the 18th International Conference on Software Engineering*, IEEE Computer Society, pp. 442-449.

Berndtsson, M., Mellin, J., and H¨ogberg, U. 1999. Visualization of the composite event detection process. In *International Workshop on User Interfaces to Data Intensive Systems (UIDIS)*, Edinburgh, September. IEEE Computer Society.

Booth, H.D. 1992. An overview of Red-Black and Finger Trees, chapter 2 of Some Fast Algorithms on Graphs and Trees, Technical Report: CS-TR-296-90, Princeton University.

Bovet, D.P. and Cesati, M. 2003. *Understanding Linux Kernel*. 3rd edition, Oreilly publishers.

Buchmann, A.P., Zimmermann,J., Blakeley,J.A., and Wells,D.L. 1995. Building an integrated active OODBMS: Requirements, architecture, and design decisions. In *Data Engineering*.

Burns, A. & Wellings, A. 2001. *Real-Time Systems and Programming Languages* (3 ed.), Harlow, England: Addison-Wesley

Bækgaard, L. and Godskesen, J.C. 1998. Real-time event control in active databases. *Journal of Systems and Software*, 42(3), August, pp.263–271.

Chakravarthy, S., Blaustein, B., Buchmann, A.P., Carey, M., Dayal, U., Goldhirsch, D., Hsu, M., Jauhuri, R., Ladin, R., Livny, M., McCarthy, D., McKee, R., and Rosenthal, A. 1989. HiPAC: A research project in active time-constrained database management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, July.

Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim, S.K. 1994. Composite events for active database: Semantics, contexts, and detection. In *20th Int'l Conf. on Very Large Databases, Santiago, Chile, September*, pp. 606–61.

Chakravarthy, S., and Mishra, D. 1994. Snoop: An event specification language for active databases. *Knowledge and Data Engineering*, 13(3), October.

Cormen, T.H., Leiserson, C.E. and Rivest R.L. 1993. *Introduction to Algorithms*. The MIT Press Cambridge, Massachusetts London, England.

Czech, J.Z., Havas, G. and Majewski, B.S. 1992. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5), pp. 257-264.

Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, S., Hsu, M., Ladin, R., McCarty, D., Rosenthal, A., Sarin, S., Carey, M.J., Livny, M., and Jauharu, R. 1998. The HiPAC project: Combining active databases and timing constraints. *ACM Sigmod Record*, 17(1), March.

Dousson, C., Gaborit, P., and Ghallab, M. 1993. Situation recognition: Representation and algorithms. In *Proceedings of the Thirteenth Int'l Joint Conference on Artificial Intelligence [IJCAI 93]*, pp.166–172.

Galton, A., and Augusto, J.C. 2001. Two approaches to event composition. Technical Report 401, Department of Computer Science., University of Exeter, 2001.

Gatziu, S., and Dittrich, K.R. 1992. SAMOS: An active object-oriented database system. *IEEE Data Engineering, Special issue on active databases*, 15(1-4), December, pp.23–26.

Gatziu, S. 1994. *Events in an Active Object-Oriented Database System*. PhD thesis, University of Zurich, Switzerland.

Gatziu, S., Geppert, A., and Dittrich, K.R. 1994. The SAMOS active DBMS prototype. Technical report 94.16, Informatik der Universit¨at Z¨urich.

Gehani, N.H., Jagadish, H.V., and Schmueli, O. 1993. COMPOSE – A system for composite event specification and detection. In *Advanced Database Concepts and Research Issues*. Springer-Verlag.

Geppert, A., Berndtsson, M., Lieuwen, D., and Roncancio, C. 1998. Performance evaluation of object-oriented active database management systems using the BEAST benchmark. *Theory and practiceof object systems*, 4(4), pp.1–16

Guibas, L.J. and Sedgewick, R. 1978. A diochromatic framework for balanced trees. *In proceedings of the 19th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, pp.8-21.

Ho, Y. 1994. *Application of Minimal Perfect Hashing in Main Memory Indexing*. Master of Engineering thesis, M. I. T. Department of EECS.

Knuth, D. 1973. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*, Addison-Wesley.

Lehman, T.J. and Carey, M.J. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan, pp. 294-303.

Litwin, W. 1980. Linear Hashing: A New Tool For and Table Addressing. *In Proceedings of the Sixth International Conference on Very Large Data Bases*, Montreal,Canada.

Mellin. J. 2004. *Resource-Predictable and Efficient Monitoring of Events*. PhD thesis, Department of Computer Science, University of Linköping, Dissertation No. 876, Sweden.

Meyers, S. 2001. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley Professional Computing Series.

Mok, A.K., and Liu, G. 1997a. Early detection of timing constraint violation at runtime. In *18th IEEE Real-time system symposium (RTSS'97)*.

Mok, A.K., and Liu, G. 1997b Efficient runtime monitoring of timing constraints. In *Proceedings of RTAS'97*.

Motakis, I., and Zaniolo, C. 1997. Formal semantics for composite temporal events in active database rules. *Journal of Systems Integration*, 7(3/4), pp. 291–325.

Ramakrishna, M.V. 2003. *Database Management Systems*. Editorial McGraw-Hill, ISBN: 0071230572, 3rd edition.

Schroeder, B.A. 1995. On-line monitoring: A tutorial. *Computer*, 28(6), June, pp.72–78.

Sleator D.D and Tarjan R.E. 1985. Self-adjusting Binary Search Trees, *JACM*, 32(3), pp 652-686.

Snapp, S.R., Brentano, J. et al. 1991. DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and an Early Prototype. In *Proceedings of the 14th National Computer Security Conference*. Washington. DC. October. pp. 167-176.

Tarjan, R.E. and Van Wyk, C.J. 1988. An O(nloglogn)-time algorithm for triangulating a simple polygon. *SIAM J. Comput.*, 17, pp. 143-178.

Zimmer, D., and Unland, R. 1999. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering*, IEEE Computer Society Press, pp 392–399.

Appendix A

The derivation for the composition cost function for chronicle context along with the monitoring cost is derived in this section.

The general simplified recurrence relation from Eq. 4.3 on p.22 is given below:

$s_{\hat{e}} - s_{\hat{e}-1} = kf(k) + k(\, f(k) + \text{context}(i, t_{e-1})\,(1+ f(k))\,)$ iff safe pointers

In chronicle, recent and continuous event contexts, the closed form functions can be derived from these non-homogeneous difference equations by substituting e with n and s with function q(n) where $q(n)=n^k \Sigma^n_{i=0} c_i * n^i$ is a trial function. For safe pointers, the result of this substitution is Eq. 11.1

$$q(n) - q(n-1) = kf(k) + k(\, f(k) + \text{context}(i, t_{e-1})\,(1+ f(k))\,) \quad \rightarrow (11.1)$$

By setting $q(n)= c_o + c_1 n$, Eq. 11.2 can be obtained.

$$c_o + c_1 n - (c_o + c_1(n-1)) = kf(k) + k(\, f(k) + \text{context}(i, t_{e-1})\,(1+ f(k))\,) \quad \rightarrow (11.2)$$

By treating $c' = \text{context}(i, t_{e-1})$ as a constant with respect to n for the moment, Eq. 11.3 is the result and defines the constant c_1.

$$c_1 = kf(k) + k(\, f(k) + \text{context}(i, t_{e-1})\,(1+ f(k))\,) \quad \rightarrow (11.3)$$

After the composition function is defined, we address the cost function of event contexts.

The closed function is:

$$q(n)= c_0 + kf(k)(2+ c')n \text{ , where } q(0) = c_0 \quad \rightarrow (11.4)$$

In this case, q(0) = 0 and the closed form function is derived as shown in the Eq. 11.5,

$$q(n) = kf(k)(2+ c')n. \quad \rightarrow (11.5)$$

Since composition(g, e, m, k, p) = s_e = q(e), according to Def. 4.4 on p. 21 and the fact that s_n = q(n), then

$$\text{composition}(g, e, m, k, p) = 2e\, kf(k) + c' e\, kf(k) \quad \rightarrow (11.6)$$

The complexity class of safe pointers and multiple terminators is derived as follows. For chronicle context $c' = t_{e-1}$ (from Def. 4.9 on p. 22). On substituting c' in the Eq. 11.6 the Eq. 11.7 is obtained.

$$\text{composition}(g, e, m, k, p) = 2e\, kf(k) + t_{e-1} e\, kf(k) \quad \rightarrow (11.7)$$

Due to Lemma 9.1 on p. 203 of Mellin (2004) t_{e-1} can be substituted with g. Therefore, for multiple terminators, Eq. 11.7 yields to the Eq. 11.8.

$$\text{composition}(g, e, m, k, p) = 2e\, kf(k) + t_{e-1} e\, kf(k) \quad \rightarrow (11.8)$$
$$= O(gekf(k))$$

For a single occurrence of the terminator, then $t_{e-1} = t_0 = 1$ that gives Eq. 11.9 derived from Eq. 11.8.

$$\text{composition}(g, e, m, k, p) = 2e\, kf(k) + e\, kf(k) = O(e\, kf(k)) \quad \rightarrow (11.9)$$

The total cost function for chronicle context for multiple terminators is derived by replacing lookup (in Def.4.4 on p. 21), composition (in Eq. 11.8), and prepare_delivery (in Def. 4.10 on p. 22) cost functions in monitor (Def. 4.2 on p. 21) giving Eq.11.10 for processing multiple terminator occurrences.

$$\text{monitor}(g, e, m, k, p) = O(g\text{f}(m)\text{f}(k)) + O(gek\text{f}(k)) + O(gep)$$
$$= O\mathbf{(}g(\text{f}(m)\text{f}(k) + epk\text{f}(k))\mathbf{)} \quad \rightarrow 11.10$$

For processing a single terminator occurrence, Eq. 11.11 is derived.

$$\text{monitor}(g, e, m, k, p) = O(\text{f}(m)\text{f}(k)) + O(e\ k\text{f}(k)) + O(ep)$$
$$= O\mathbf{(}\text{f}(m)\text{f}(k) + epk\text{f}(k)\mathbf{)} \quad \rightarrow 11.11$$

In a similar way, the complexity classes for other event contexts are also derived.

Appendix B

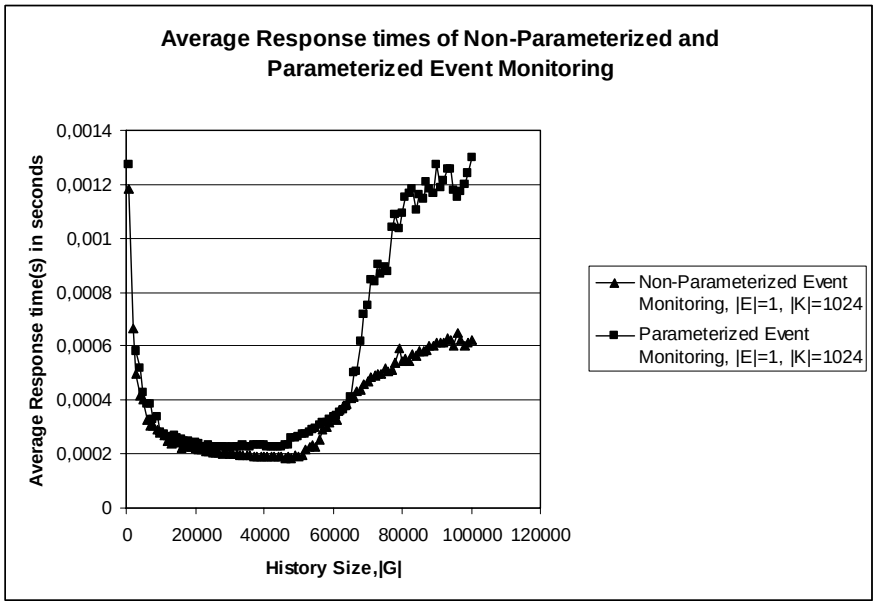

Fig.B.1: Average response times of parameterized and non-parameterized event monitoring

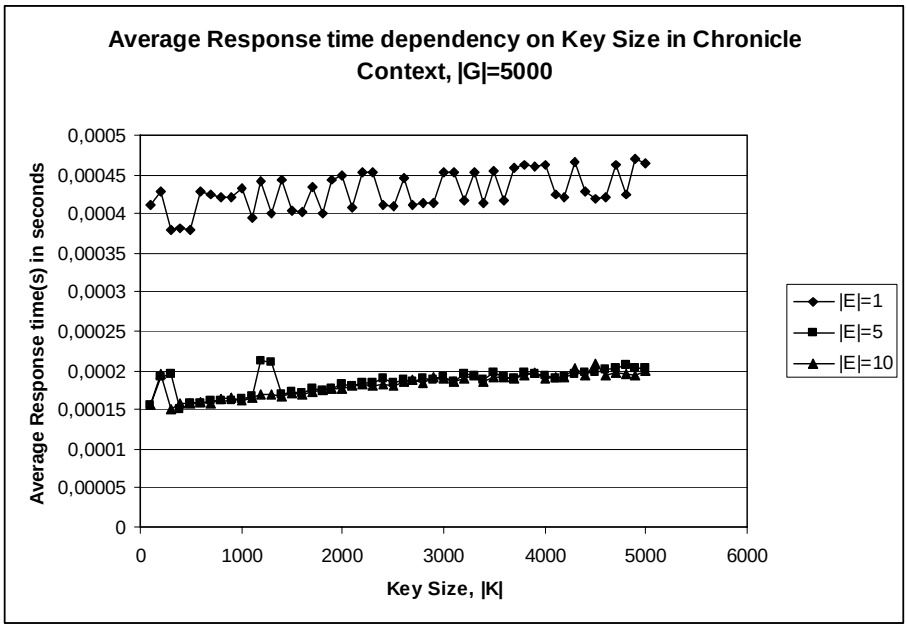

Fig B.2: Response time dependency on Key Size in chronicle context