# Chapter 7

# Efficient Event Filtering

*Efficiency is intelligent laziness.*
*David Dunham*

High frequencies of events and large numbers of providers and clients at the ENS demand efficient processing of events (cf. Requirement R6 in Chapter 2). In this chapter, we address performance of event notification systems in three steps: First, we use a performance model to evaluate parameters that influence the event processing time[1] of a service. In particular, we study the service's event processing time by model simulations. We show that the filter part has major influence on the overall processing time of an event notification system. Consequently, we focus on efficient algorithms for event filtering.

In the second step, filter methods for primitive events are analyzed: Typically, performance evaluations of ENS rely on test sets of equally distributed profiles and events. Our study of different scenarios shows that the assumption of equally distributed data does not hold for typical ENS applications. For example, profiles for a facility management system typically define an interest in a small range of values of data of high importance, e.g., failure notifications. Event data are recorded by sensors, they are often normally distributed (Gauss). We propose an adaptable filter algorithm that optimizes the profile tree for certain applications based on the data distributions. We introduce both an event-centric and a client-centric approach.

After having studied efficient filtering of primitive events, the third part of the chapter concentrates on the filtering of composite events. The filter time for a composite event is the time between the occurrence of the last event contributing to a composite event and the client notification. To the best

[1]We do not additionally focus on the throughput of the service, because it can be derived from the event processing time. Moreover, for event notification services, the event processing time is the most relevant measure. It estimates the time between the occurrence of an event and the notification of a client.

| Parameter | Values |
|---|---|
| Application-based | |
| event frequency per provider | 0.0003 to 2 events / sec at one sensor |
| total event frequency | 10 to 2,000 events /sec |
| attribute number per event | 4 to 10 per provider |
| overall attribute number | 40 to 50 in one application |
| attribute domain size | 15 different domains, from simple on/off to 4 Byte floating point numbers |
| profile number | 200 to 20,000 |
| provider number | 1,000 - 20,000 sensors |
| event message size | about 20 Byte |
| Structural parameters | |
| observation method | varying |
| persistency of event messages and notifications | rarely, only for selected event types |
| Technical parameters | |
| bandwidth of connections | 9600 Baud (about 20-30 messages / sec effective bandwidth) |

Table 7.1: Typical parameter values for a mobile maintenance service (Example 7.1)

of our knowledge, all existing methods for composite event detection consist of two steps: In the first step, the primitive profiles are evaluated and in the second step, the corresponding composite profiles are identified. Thus, the composite event is detected in a separate step after the filtering of the contributing primitive events. In this chapter, we propose a new method for the filtering of composite events that integrates the detection of composite events into the detection of primitive events: During the filtering of a primitive event, its contribution to a composite event is tested. In that way, the composite event is detected successively. No additional step is required for the identification of the composite event after the last contributing primitive event has been detected. The identification of the composite event after the last contributing primitive event is accelerated and, additionally, the overall filtering time is reduced for primitive and composite events.

## 7.1 Performance Model

To identify the influence of the system parts (as introduced in Chapter 3) on the system's performance, we introduce a Petri Net model. The model focuses on the dynamic interplay of the system parts within an ENS server. The term *ENS server* refers to an ENS that acts within a network of servers.[2] Yan and Garcia-Molina [YGM94a] introduced a model that focuses on scalability issues in a distributed system. In their approach, each ENS server is modelled as a queuing network[3]. In contrast, our model is more fine grained and considers the internal parts of each ENS server, which allows for a more detailed performance analysis. In addition, our model involves parts that cannot be modelled using queuing networks, such as scheduled observation. Therefore, our approach is more appropriate for the modelling of an integrating service that covers various sources. The efficiency of the ENS server is measured in

[2] In this thesis, we focus on performance within an ENS server. We recognize the usefulness of the approach to reaching scalability by filter distribution, which has been extensively addressed in research, e.g., [Car98, MFB02, PB02]. For our prototypical implementation, we employ the techniques developed there.

[3] For a general introduction into modelling using queuing networks see, e.g., [LZGS84]

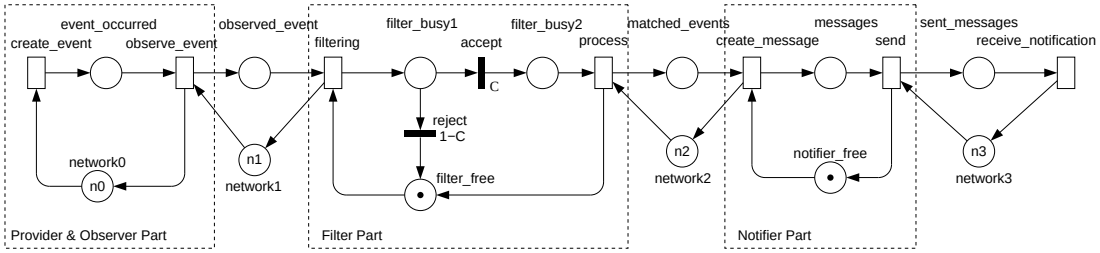



Figure 7.1: Basic performance model for event notification services

event processing time, i.e., the time from event occurrence to client notification. First, we introduce a basic model. Then, we enhance the observer part to cover variants of active and passive observations. Other model extensions are discussed briefly.

To illustrate our model, we use an example from Scenario 1: Facility Management. We concentrate on a mobile maintenance service that is one application within a facility management system. The example is briefly introduced here and extended throughout the chapter.

***Example 7.1 (Facility Management: Mobile Maintenance Service)***
*We consider a medium commercial building with about 1000 actors/sensors sending event messages to the ENS. The system uses a personal computer, the service communicates via the EIB bus (see Scenario 1 on Page 12). The service filters incoming messages from sensors.*

*Table 7.1 shows parameter values for a mobile maintenance example covering one building. Usually, up to five buildings are covered by one service. The values given in the table range from current conditions to future system requirements. We distinguish application-dependent, structural, and technical parameters. The application-dependent parameters have been derived from the scenarios in Chapter 2. The structural parameter 'observation method' (active vs. passive, scheduled vs. event-based) has already been addressed in Chapter 6. Only one technical parameter is considered: the 'bandwidth of connections'.*

Figure 7.1 shows our basic model. We use a Generalized Stochastic Petri Net (GSPN) [MBC+95]. The structure of a Petri Net (PN) is a bipartite graph consisting of places, transitions, and directed arcs. The places model states; they may contain tokens. Transitions represent activities or changes of states. Places and transitions are connected by the directed arcs. Transitions can fire if all its input places contain at least one token. Generalized Stochastic Petri Nets contain timed transitions with associated (random) firing time, e.g., exponential, deterministic and immediate transitions.

Following our reference model in Chapter 3, the event information in our Petri Net model flows from left to right. As known from the previous chapters, the communication modes of provider and observer are strongly related. The two parts are therefore modelled together in the *provider & observer part*. Events occur (modelled by the transition `create_event`) and are observed (transition `observe_event`). All transitions are modelled, if not explicitly stated differently, by transitions with exponential distributions in order to describe the independence of the events. Each token in state `event_ocurred` symbolizes an event that occurred but has not been observed yet. The tokens in `network0` and `network1` model the limited bandwidth of the network connections between provider and observer. The number of tokens `n1` models the maximal number of events that can be sent over that

network connection. Tokens in state `observed_events` represent event (messages) that have been observed and are queued to be filtered according to the client profiles.

The *filter part* in the second box has two stable states: being `free` to work on new messages or `filter_busy` while matching the event messages and the profiles. The incoming event messages are divided into two groups, modelled by the immediate transactions `accept` and `reject`. The ratio of the transitions' priorities model the ratio of matched events, defined by the coverage $C$, to the unmatched events $(1-C)$. The filter time is subdivided in two transitions (`filtering` and `process`) supporting the modelling of filtering mechanisms that depend on both the number of events and the number of matched events in different proportions. Tokens in state `matched_events` represent events that have been matched by at least one profile within the system. These are then sent to the notifier via network connection bound by bandwidth `n2` in `network2`.

The *notifier part* in the third box describes the sending of notifications about matched events. The transition `create_message` models the creation of each message, while `send` symbolizes the actual sending of each message. If more than one event is reported in each message, the average number of events per message can be included in the model by defining a multiplicity factor for the arc between `matched_events` and `create_message`. This is also a simple approach to model the assembly of primitive events to form composite events. Tokens in state `sent_messages` represent the messages sent for clients' notifications, `n3` in `network3` models the bandwidth of the network connection to the client. The client is modelled by the transition `receive_notification` as a simple token sink; the messages are processed as they enter the client's site.

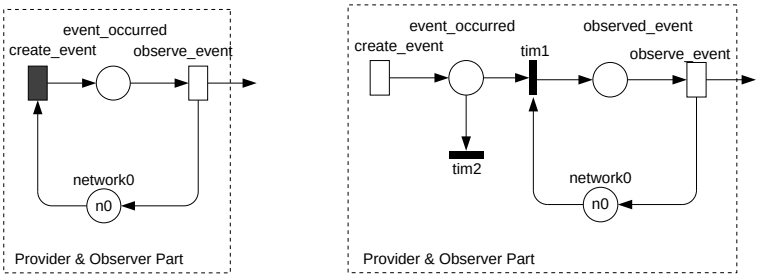
The external network traffic in the four connections can be modelled by the definition of multiplicity factors for all four arcs. For example, for network load of 10:1 external to event messages, all arcs would have the multiplicity of 11 (10 external messages + 1 event message).

To model dependencies on the number of profiles and profile distributions, we introduce the concept of *coverage*. Recall from Chapter 3 that the event space $\mathbb{E}$ is the set of all possible events known to a certain system; the profile set is denoted $\mathbb{P}$.

***Definition 7.1 (Profile Coverage)***
*The profile coverage $C$ is the probability of a set of profiles to be matched by a random event of the event space $\mathbb{E}$. It is defined by $C = \frac{|\mathbb{E}|}{|\mathbb{P}|}$, where $|\mathbb{E}|$ and $|\mathbb{P}|$ denote the size of the event space and the profile set, respectively.*

The sizes of event space and profile set are defined by the domains of the attributes in events and the fraction of these domains covered by profiles.

***Example 7.2 (Profile Coverage in Facility Management Services)***
*For facility management, about 20,000 profiles are defined for each building. The profiles range from small portions covering a specific sensor to broader ones covering general situations. The overall coverage for the profiles is about 80 percent. For our mobile maintenance service (cf. Example 7.1) as one application within the facility management system, each profile covers a small portion of the event space, e.g., certain alert situations. The profiles are mostly distinct. The overall coverage for 1000 profiles is about 1 percent.*




(a) Deterministic Provider (b) With data loss at overload

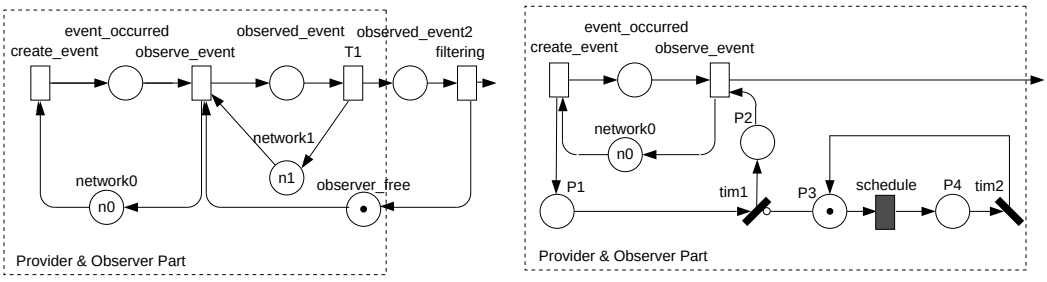
Figure 7.2: PN models of active providers with passive observers

**Model Analysis.** A static analysis identifies the mean event processing time $E[T]$ as

$$E[T] = E[T_{obs}] + E[T_{network1}] + C\big(E[T_{filter}] + E[T_{notif}] + E[T_{network2}]\big) \tag{7.1}$$

The influence of the internal network $n_2$ is negligible and not shown in the equation. The processing times of observer $E[T_{obs}]$, filter $E[T_{filter}]$, and notifier $E[T_{notif}]$ depend on their respective frequencies and that of the subsequent parts. For example, the observer processing time $E[T_{obs}]$ depends on the provider and observer frequency $E[T_{obs}] = \frac{1}{\lambda_{prov} - \lambda_{obs}}$. The network access time depends on the network loads $L_1$ and $L_3$, respectively, and on the probability of data loss due to network overload $(1 - \pi_{n1})$: $E[T_{network1}] = \frac{L_1}{(1-\pi_{n1})\lambda_{obs}}$. The coverage $C$ depends on the distribution of attribute values in profiles and events, and the profile number $C = C(P_e, P_p, p)$. The system throughput is then calculated by $1/E[T]$.

We now discuss model variants to cover active and passive observation. In the basic model, the provider sends independent event messages to a minimal (passive) observer that acts as port to the service. Providers that supply scheduled sensor readings to a passive observer can be modelled as deterministic transitions (see Figure 7.2(a)). In Figure 7.2(b), we enhance the basic observer such that a data loss at overload is modelled. If the event frequency at the observer is too high, or reaches single peaks, the system does not block but data is lost (tim1 and tim2 control the data loss). An active observer queries the provider's site over the network, either triggered by events (Figure 7.3(a)) or based on a




(a) Event-triggered Observer (b) Scheduled Observer

Figure 7.3: PN models of passive providers with active observers

| Parameter | Description |
|---|---|
| $\lambda_{prov} = 10 - 2,000\, s^{-1}$ | mean event generation rate (events/sec) |
| $\lambda_{observeevent} = 20,000\, s^{-1}$ | mean event observation rate (observed events/sec) |
| $\lambda_{filtering} = 50 - 10,000\, s^{-1}$ | mean filtering rate (events/sec) |
| $\lambda_{process} = 1000\, s^{-1}$ | auxiliary: mean forwarding rate for matched events (events/sec) |
| $\lambda_{createmessage} = 1000\, s^{-1}$ | mean rate for message creation (message/sec) |
| $\lambda_{send} = 1000\, s^{-1}$ | mean rate for message sending (message/sec) |
| $\lambda_{client} = 1000\, s^{-1}$ | auxiliary: mean rate of message consumption at client site |
| $n0 = 130,000,000$ | capacity at provider's site |
| $n1 = 130,000$ | network capacity between provider and service |
| $L = 10$ | network traffic noise (noise/event) |
| $accepted = 1$ | proportional constant for matched events |
| $rejected = 100$ | proportional constant for unmatched events |
| $n2 = 650,000$ | network capacity between filter and notifier |
| $n3 = 1,000,000,000$ | network capacity between service and client |

Table 7.2: Summary of model parameters for a mobile maintenance service

schedule (Figure 7.3(b))[4]. In the latter case, the observation is prevented by the inhibitor arc until scheduled (based on the deterministic transition `schedule`). The observer transition is modelled as infinite server in order to collect all events at the provider's site.

The model may be refined at several parts: the filter, notifier, persistency and network and traffic delays. We discuss these refinements only briefly. The filter part in the basic model is only a simple approximation to the filtering process where influencing parameters are subsumed in the coverage $C$. A thorough extension would require the employment of Colored Petri Nets (CPN). We discuss the filter influences in detail in Section 7.2. Variations of the notifier part, e.g., active and passive delivery of notifications may be modelled very similar to our modelling of the provider & observer part. Additional network delays can be modelled using deterministic transitions with a frequency depending on the network's bandwidth. The external network traffic can be modelled by definition of multiplicity factors for the connecting arcs.

We show results for a parameterized simulation of the model for our facility management example. For the model simulation, we use the TimeNET Tool [ZFGH00] developed at the Technical University Berlin that supports Generalized Stochastic Petri Nets[5].

***Example 7.3 (Model Evaluation for a Mobile Maintenance Service)***

*The parameter settings for the model are shown in Table 7.2. They are based on the application parameters introduced in Table 7.1. We tested the influence of varying event and filter frequencies and, thus, varying installation sizes. From the algebraic analysis in Equation 7.1, the asymptotic progression of the processing time curves can be derived. Here, we focus on specific data for our example. Four simulations $A$–$D$ are discussed.*

A) Simulation: Event Processing Time Depending on Provider Frequency. *In this first evaluation, the event processing time is measured depending on the providers' event frequency, i.e., the event generation rate at the providers. To emphasize the opposition to the ENS filter frequency, we call this parameter 'provider frequency' – it is modelled by the provider transition. We use a parameter setting for a typical*

[4] These observation modes have been introduced and described in Chapter 3 at Page 36 and Table 3.1.

[5] All simulations were performed until a 95 percent confidence level was reached.

*implementation of the maintenance service under* current conditions*. The provider transition models* $40$–$200$ *actors sending events every 2 seconds. The parameters for provider and filter frequency are:* $\lambda_{prov} = 10 - 50s^{-1}$*,* $\lambda_{filter} = 50s^{-1}$*. The results are shown in Figure 7.4. The provider frequency is bound by the filter frequency, otherwise the tailback of the event messages blocks the service and queuing-time grows to infinity. Already this very basic example shows why many EIB-Implementations have serious performance problems [Sch03a] when using a large number of actors connected to the bus (as allowed according to the EIB standard).*

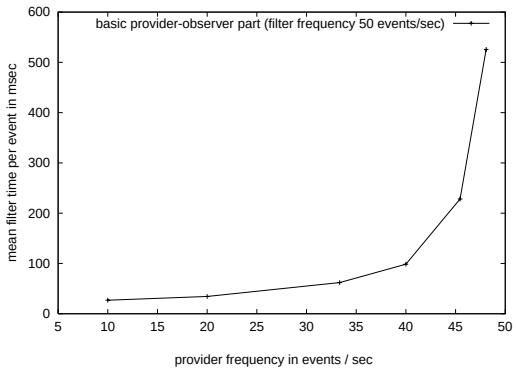



Figure 7.4: Simulation results: filter time depending on provider frequency

B) Simulation: Event Processing Time Depending on Filter Frequency. *In the second evaluation, event processing time is measured depending on the filter frequency. We use a parameter setting for a typical implementation of the maintenance service under* future conditions*; the service provides enhanced functionality for facility management purposes [Gra02]. The filter transition models* $20,000$ *actors sending events in* $10s$ *intervals. The parameters for provider and filter frequency are:* $\lambda_{prov} = 2000s^{-1}$*,* $\lambda_{filter} = 2,000 - 10,000s^{-1}$*. The results are shown in Figure 7.5. As expected, the filter frequency is bound by the provider frequency, otherwise the system blocks and the processing time surges. In this example, the filter part has to provide a minimal filter rate of 4000 filtered events per second to provide sufficient performance. Consequently, a system has to offer a filter time of* $0.25msec$ *per event for* $20,000$ *profiles.*

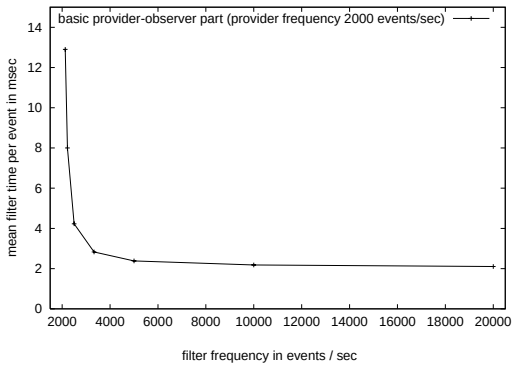
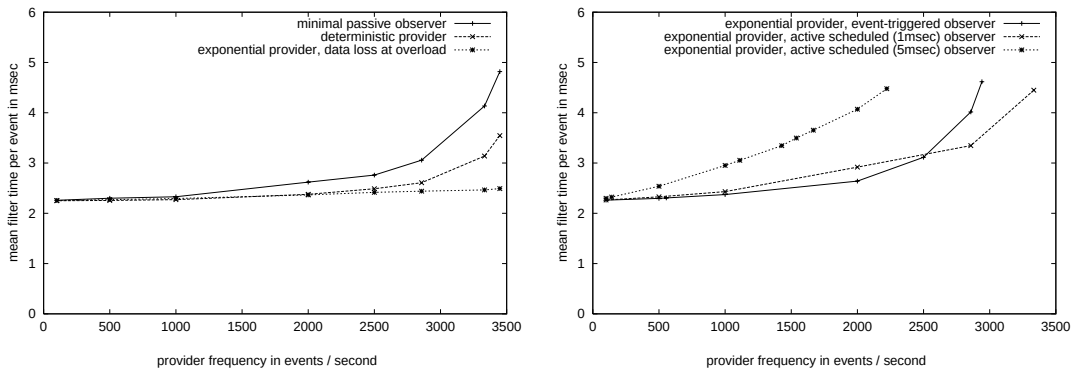



Figure 7.5: Simulation results: filter time depending on filter frequency




(a) Passive Observation (b) Active Observation

Figure 7.6: Simulation results: filter time depending on observer variants

C) Simulation: Event Processing Time Depending on Observation Methods. *Figure 7.6 shows the system's processing times for varying observation methods. In our facility management example, different observation methods are applicable. We simulated the system behavior using six different provider & observer parts. Figure 7.6(a) shows the results for passive observation depending on the provider frequency. We compare the mean filter times for minimal passive observer, deterministic provider, and data loss at overload. The respective PN models have been shown in figures 7.1, 7.2(a), 7.2(b). The events sent on a regular basis by a deterministic provider result in shorter event processing time compared to independently occurring events (exponential transition in minimal provider) when using the same transition firing frequency. The third variant shows a provider with data loss at overload: The system does not block when overloaded. Figure 7.6(b) shows the results for active observer variants: an event-triggered observer and an active scheduled observer with two different observation schedules. The respective PN models have been shown in figures 7.3(a) and 7.3(b). As discussed in Chapter 6, all three active observation variants are slower than the passive observation with active provider. In the direct approach, the curve bends differently than in the two scheduled ones. The curves cross at a provider frequency of* $2500$ *events per second. The reason is that the direct method is triggered by the events; therefore, the provider's influence is stronger.*

D) Simulation: Event Processing Time Depending on Bandwidth. *Figure 7.7 shows the results of our simulation of changing bandwidth between provider and service. The event processing time has been measured for current and future conditions for provider and filter frequencies. For current systems, a minimal bandwidth of 5000 events per second is required (= about 1MBit/sec effective bandwidth). Consequently, an EIB bus system with maximal 30 events per second as described before is not suitable. At least Ethernet connections are required. For future systems, a bandwidth of at least* $30,000$ *events per second (= about 6 MBit/sec effective bandwidth) is needed: Ethernet and Wireless LAN (more than 10 MBit/sec) are suitable.*

Our basic model gives a first impression of the event processing times of a modelled system. Equation 7.1 describes the event processing time depending on system parameters. Because several of these parameters are determined for a certain scenario (e.g., provider frequency, coverage, network parameters), the number of open parameters to enhance the service's performance is limited. The following

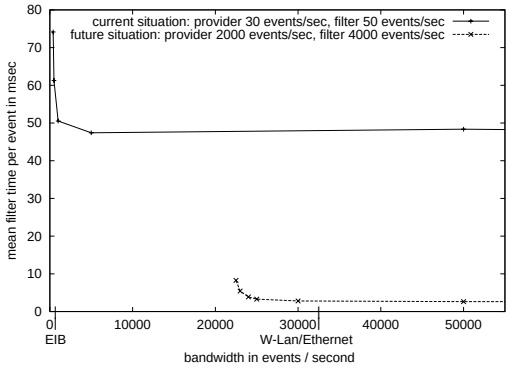



Figure 7.7: Simulation results: filter time depending on network bandwidth; The effective bandwidth of EIB, wireless LAN and Ethernet have been included as references.

parameters are application independent: the time for filtering, message creation, and sending. Out of these, the most dominating part of the system's processing time is created by the filtering method. Therefore, the efficiency of filter algorithms is of major importance for the system's performance. In the next sections, we focus on filter algorithms for primitive and composite events.

## 7.2 Distribution-driven Primitive-Event Filtering

In this section, we propose an algorithm for efficient filtering of primitive events. The algorithm uses information about the distributions of attribute values in profiles and events to achieve best performance. First, we argue that the performance of filter algorithms depends on the distributions of attribute values in event messages and profiles. For simplicity, we refer to these value distributions for each attribute as *distributions of events and profiles*. Then, we propose a model for the fastest filter algorithm that mirrors the algorithm's dependence on event and profile distributions. Finally, we propose an enhanced algorithm to provide best average-case performance.

Three basic filtering approaches can be distinguished: naive filtering, clustered filtering, and tree-based filtering [FLPS00, Bor01]. The basic unit for event filtering is the time to filter one event attribute value against one profile predicate. We assume that for a certain event message type, for each attribute one and only one predicate is defined in each profile. We estimate the number of filter steps $S$ as follows: In the naive approach, all profiles are tested successively and $S = p * a$, where $p$ is the number of profiles and $a$ the number of attributes in events and profiles. In an enhanced naive approach (e.g., in Xyleme [NACM00]), all distinct predicates are tested separately. The number of distinct predicates $pr$ depends on the profile distribution $P_p$, the event distribution $P_e$, and the number of profiles $p$: $S = pr(P_p, p, P_e) * a$. In the clustered approach (e.g., in Conquer [LPTH99] and NiagaraCQ [CDTW00]), the predicates are clustered according to the operators used. Then, the most selective predicates are tested first. In that way, similar predicates can be matched faster and $S = pr'(P_p, p, P_e) * a$.

For the tree-based approach (e.g., in [ASS+99, GS95, HCKW90]), the predicates are ordered in a profile tree according to the attributes they refer to. For each attribute, a selection of predicates has to be tested using binary search. It follows that $S = \log_2 pr(P_p, p, P_e) * a$. As seen from this argumentation

and according to [FLPS00, Bor01], the tree-based filter algorithms show the best performance results.

Following our argumentation, the number of filtering steps depends on the profile and event distributions. This influence has been evaluated by a test implementation [Bor01]. From this work, we show selected results for the filter time per event when using the tree-algorithm (see Figure 7.8).[6] Two observations can be made. First, the smaller the domain size the faster is the algorithm. The reason is the influence of the attribute domain sizes on the overlap of profiles: With higher profile overlap less distinct predicates have to be tested. Second, the distribution of the values within each attribute of the profiles has significant influence on the filter performance. This can be seen in Figure 7.8 for the results regarding domain size 10,000. For the smaller domain size, the first effect has stronger influence. Again, the second effect is due to the changing profile overlap and the number of distinct predicates. Both effects vanish when the profile set completely covers the event space. For uniform distributions, this depends only on the number of profiles; for other profile distributions, the event space may never be completely covered.

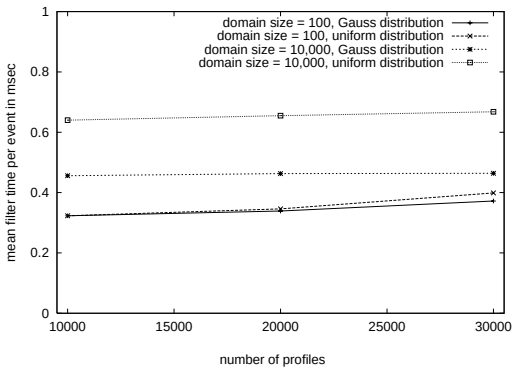



Figure 7.8: Filter time of tree-based algorithm for different domain sizes and value distributions

We have argued that the distribution of events and profiles has impact on the filter performance. We now focus on tree-based filtering under different event and profile distributions: First, we model a tree-based filter algorithm such that its distribution dependency is made explicit. Then, we propose a new version of the algorithm that uses the information about the distributions to achieve better performance.

### 7.2.1 Distribution-dependent Filter Model for Tree Algorithm

In this section, we introduce a distribution-dependent model of tree-based filter algorithms. Our model is based on the fastest tree algorithm for (attribute;value) pairs introduced in [GS95]: From a given set of profiles, a deterministic finite state automaton (DFSA) is created. This automaton is translated into a profile tree. To describe the tree and to model the algorithm, we introduce a notation for attributes and their domains in event messages and profiles: For a given application, we consider a fixed set $A$ of attributes $a_j$, with values belonging to given domains $D_j$. The number of attributes in events and profiles is $n$, $j \in [1, n]$. We denote the domain-size with $d_j \in \mathbb{N}$. For readability reasons, we use $D$ instead of the correct $D_j$ in general cases.

The number of profiles is denoted $p$. Considering profiles for value or range tests, each attribute's domain $D$ is divided in, at most, $(2p-1)$ subsets (referred to in the profiles) and an additional subset $D_0$ which is not referred to in any profile (see Figure 7.9). Note that inequality tests can be translated to

[6] Unfortunately, no data is available for a higher number of profiles. Test parameters: 1000 uniformly distributed events, 3 attributes in events and profiles, equality operators in profile predicates. The algorithms have been implemented with Java, the tests have been performed on a personal computer with 350 MHz and 64 MB RAM.

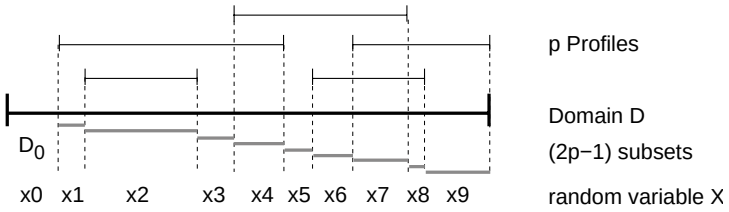



Figure 7.9: Division of attribute domain $D$ in $(2p-1)$ subsets

range tests. The $(2p-1)$ subsets are based on sets of non-overlapping subranges created from the at most $p$ different ranges defined in the $p$ profiles [GS95]. In routing applications, each observed event is matched by at least one profile, and therefore $D_0 = \emptyset$. For filtering applications holds $D_0 \neq \emptyset$. We define $\overline{D} := D \setminus D_0$. The filtering is finally based on a profile tree of height $n$. The leaf nodes of a the tree refer to the client profiles. Each non-leaf level $j \in [1, n]$ of the tree corresponds to the respective attribute $a_j$ in the profiles. Branches starting at node $a_j$ at level $j$ are labelled with disjunct predicates regarding attribute $a_j$ as defined in the profile set. The filter algorithm searches a *single path* to the matching profiles via the attribute predicates. For an event that matches one or more profiles, there is only a single path to follow for finding matching profiles.

**Algorithm 7.1** Tree-based algorithm [GS95]: sequential tree-search for single matching path

**input**: profile tree with attribute levels $a_1 ... a_n$ and branches $b_1 ... b_m$
event-message with (attribute;value) pairs
**output**: list of matching profiles or NULL

```
1:  current-node := root                                   /* root = a_1 */
2:  while (current-node <> leaf-node) do
3:      branch-to-follow := NULL
4:      iterate sequentially through branches i ∈ [1, m]
5:          if branch predicate at b_i true for event-message
6:          then branch-to-follow := b_i
7:      if (branch-to-follow <> NULL)
8       then follow the branch to the next node: current-node := branch-to-follow → a_{i+j}
9:      else EXIT
10: output current-node
```

The basic algorithm, which uses sequential search at each node, is shown in Algorithm 7.1. If using binary search at each node, the filter time of a tree algorithm for range tests is bound by $O(n \log p)$ as shown in [FLPS00].The following example from facility management serves as an illustrative reference:

***Example 7.4 (Tree-based Filtering)***
*Given is a facility management system for environmental monitoring of laboratories: The service delivers sensor readings of temperature, humidity, and radiation. The attributes and their domains are:*

$$a_1: \quad D_1 = [-30, 50] \ (temperature\ in\ ^\circ C)$$
$$a_2: \quad D_2 = [0, 100] \quad (humidity\ in\ \%)$$
$$a_3: \quad D_3 = [1, 100] \quad (UVA - radiation\ in\ mW/m^2)$$

*In the following profiles, $*$ denotes the fact that the client did not specify this attribute (*don't care *value).*

$$P1 : profile(a1 \geq 35, a2 \geq 90, a3 = *)$$
$$P2 : profile(a1 \geq 30, a2 \geq 90, a3 = *)$$
$$P3 : profile(a1 \geq 30, a2 \geq 90, a3 \in [35, 50])$$
$$P4 : profile(a1 \in [-30, -20], a2 \leq 5, a3 \in [40, 100])$$
$$P5 : profile(a1 \geq 30, a2 \geq 80, a3 = *)$$

*The profile tree is depicted in Figure 7.10. In the tree, $*$ refers to* all *possible values, $(*)$ denotes* all other *possible values of the given attribute. Note the subranges constructed from the overlapping profile ranges. Let us consider the following event:*

$$\begin{aligned} e_1 : \ event(&temperature = 30\ ^\circ C, \\ &humidity = 90\ \%, \\ &radiation = 2\ mW/m^2) \end{aligned} \tag{7.2}$$

*For this event, the filtering path in the tree follows the edges $[30, 35) \rightarrow [90, 100] \rightarrow (*)$ (marked with a thick line). The event $e_1$ is matched by the profiles $P2$ and $P5$.*

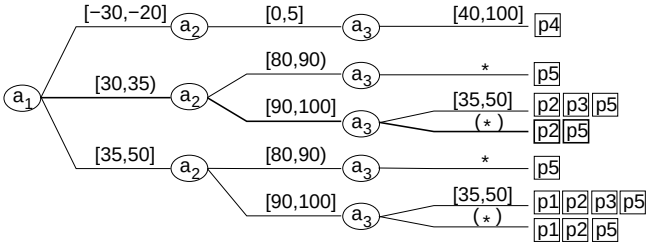



Figure 7.10: Profile tree for Example 7.4: The evaluation path for event $e_1$ as defined in Equation 7.2 is marked.

The performance of the filter algorithm is measured in comparison steps (# operations), because the structure is stored in main memory and no access to hard disc is necessary after initialization. Both events and profiles have certain probability distributions for each attribute, given as continuous density functions (for continuous values) or discrete probability values (for discrete values). The filter time is now described depending on these distributions. We start by considering a sequential search in the profile values, then we evaluate the influence of other search strategies. We assume the profile values to be naturally ordered, i.e., alphanumerically ordered according to their value.

The distribution of the (continuous) event values of a certain attribute can be reformed as a distribution of at most $(2p - 1)$ discrete values, referring to the $(2p - 1)$ subsets of $\overline{D}$. Thus, the probability of each subset is the sum of the probability values for the values in the subset for the discrete case, and the integral of the density-function over the subset for the continuous case. Each attribute value of an event is then modelled as the value of a discrete random variable $X$ within an experiment. The domain $W$ of $X$ is described by the $(2p - 1)$ subsets of $\overline{D}$. Additionally, $x_0$ refers to the subdomain $D_0$ (see Figure 7.9). The distribution of $X$ is given as $(x_i, P_e(X = x_i)), x_i \in (W \cup \{x_0\})$, where $x_i$ refers to

the numbered subsets. The distribution of the values of a given attribute within the observed events is denoted by $P_e$, the similarly defined profile distribution as $P_p$. Formally, we define the zero-subdomain $D_0 \subseteq D$ as the set of values $v \in D$, which do not occur in the profile set. The probability of these attribute values is zero:

$$D_0 = \{v | v \in D, P_p(v) = 0\}$$

The domain-size of $D_0$ is denoted $d_0$. Because $W \cup x_0$ covers $D$, it follows that $1 = \sum_i P_e(X = x_i) = \sum_i P_p(X = x_i)$. If $D_0 = \emptyset$, then the number of filter operations for each attribute $a$ can be described as the expectation for the random variable $X$. This correlation is essential for our studies of the algorithm. If $D_0 \neq \emptyset$, the performed filter operations depend heavily on the specific filter algorithm implementation.

The filter time $F(a, P_p, P_e)$ of a filter algorithm (measured in number of filter operations), where each attribute $a$ is based on given distributions for profiles and events, can be expressed as

$$F(a, P_p, P_e) := E(X) + F_0(P_e, x_0) \tag{7.3}$$

with

$$E(X) = \sum_{x_i \in W} x_i P_e(X = x_i) = \sum_{x_i \in W} x_i P_e(x_i)$$

From the equation, we conclude that the expectation $E(X)$ may vary according to the numbering order of the $x_i \in W$. We use this characteristics in the next section for our improved tree algorithm. If $D_0 \neq \emptyset$, the non-matching events have to be filtered, on average, in $F_0(P_e, x_0) = f_0 * P_e(X = x_o)$ operations, where $f_0$ refers to the number of steps needed to identify a non-matching event.

The distributions for the values of each of the $n$ attributes of an event are not independent, the notion of conditional distributions is required in this context. The number of comparison steps for each of the $n$ attributes in the profiles is then defined as the conditional expectations for $j \in [1, n]$

$$E(X_j | X_{j-1}, .., X_1) = \sum_{x_i^j \in W^j} x_i^j P_e(x_i^j | x^{j-1}, .., x^1)$$

The number of comparison steps for the profile matching of a certain event corresponds to an experiment with $n$ random variables $X_j$, $j \in [1, n]$, which are not independent. If $D_0^j = \emptyset$ for all $j$, then the overall number of steps corresponds to the expectation

$$E(\sum_j (X^j | X^{j-1}, .., X^1)) = \sum_j E(X^j | X^{j-1}, .., X^1)$$

Thus, the filter time $F$ of an algorithm for all attributes can be expressed as

$$\begin{aligned} F &:= \sum_{j=1}^{n} F(a_j, P_p^j, P_e^j) \\ &= \sum_{j=1}^{n} E(X^j | X^{j-1}, .., X^1) + \sum_{j=1}^{n} F_0(P_e^j, x_0^j) \end{aligned} \tag{7.4}$$

Equation 7.4 describes the influences on the filter time of the algorithm: The filter time depends on the attribute order in the tree, on the attribute value order at each level, and on the attribute value distributions in events and profiles. We use our model to develop an enhanced tree-algorithm that utilizes information about distributions of events and profiles to achieve best average-case performance.

### 7.2.2 Distribution-based Tree Algorithm

For the construction of our distribution-based tree algorithm, we now assume that distributions of profiles and events are known to the system. The distribution-based algorithm evaluates first those event-values and attributes that have the highest selectivity. Selectivity can be defined for each attribute value and for event-attributes, as we shall see. The profile tree is reordered such that attributes with high selectivity are at the top level of the tree, and for each attribute the values with highest selectivity are tested first.

**Value Selectivity.** For each attribute, we define a reordering of the values $x_i \in W$ as a function $o_v$ on the set of index-values $i \in [1, 2p-1]$ with $o_v : [1, 2p-1] \rightarrow [1, 2p-1]$. The expectation of X is then

$$E(X) = \sum_{x_{o_v(i)} \in W} x_{o_v(i)} P_e(x_{o_v(i)}) \tag{7.5}$$

The reordering is based on the value selectivity, a function $s_{val}$ defined on the attribute values $s_{val} : W \rightarrow \mathbb{R}$. The reordering follows the descending selectivity such that

$$\forall x_i, x_j \in W : s_{val}(x_i) \leq s_{val}(x_j) \Rightarrow o_v(i) > o_v(j)$$

The selectivity of values not contained in the profile tree is defined as zero. The order of values with equal selectivity is arbitrary (such as the alphanumerical order of the values). For the value selectivity we can define various measures, for instance, the ranked tree method [YGM94b] uses IR-like ranking information for each keyword. The binary search defines another measure, as used in [ASS+99, GS95]. It is not equally appropriate for all applications, as our test results show. Based on our discussion of equations 7.4 and 7.5, we propose three additional measures:

**V1.** Probability of the attribute values according to event distributions:
ordering of the profile values with descending $P_e(x_i)$.

**V2.** Probability of the attribute values according to profile distributions:
ordering of the profile values with descending $P_p(x_i)$.

**V3.** Probability of the attribute values according to event and profile distributions:
ordering of the profile values with descending $P_e(x_i) * P_p(x_i)$ .

The influence of the reordering on the expectation is demonstrated in the next example:

***Example 7.5 (Enhanced Tree-based Filtering using Value-based Reordering)***
*Let us assume the following probabilities for the values of attribute $a_1$(temperature) and the resulting reordering according to Measure V1.*

$$\begin{aligned}
x_1 = [-30, -20] \quad & P_e(x_1) = 2\%, \quad & o_v(1) = 2 \\
x_2 = [30, 35] \quad & P_e(x_2) = 1\%, \quad & o_v(2) = 3 \\
x_3 = (35, 50] \quad & P_e(x_3) = 80\%, \quad & o_v(3) = 1 \\
x_0 = [-20, 30] \quad & P_e(x_0) = 17\% &
\end{aligned}$$

*The expectation is*

$$\begin{aligned} E(X^1) &= \sum_{x^1_{o(i)} \in W} x^1_{o(i)} P_e(x^1_{o(i)}) \\ &= x^1_{o(1)} P_e(x^1_{o(1)}) + x^1_{o(2)} P_e(x^1_{o(2)}) + x^1_{o(3)} P_e(x^1_{o(3)}) \\ &= x^1_2 P_e(x^1_2) + x^1_3 P_e(x^1_3) + x^1_1 P_e(x^1_1) = 0.87 \end{aligned}$$

*The number of filtering operations to identify non-matching events depends on the implementation. In our prototype implementation, a non-match is discovered after the number of steps that would have been needed to identify the requested value in the tree. It follows with Equation 7.4 the filter time* $F = 1.21$*. Reordering according to binary search leads to the expectation* $E(X^1) = 1.65$*. The identification of non-matches takes* $f_0 = \log_2(2p-1)$ *steps, this leads to* $F_0 = 0.34$ *and therefore* $F = 1.99$ *in our example. In the following examples, we only discuss the expectation for successful matches.*

**Attribute Selectivity.** We define a reordering of the attributes $a_j, j \in [1,n]$ as function on the attribute indexes: $o_a : [1,n] \rightarrow [1,n]$. The expectation of the $n$ random variables $X_j, j \in [1,n]$ is then

$$E(\sum_{o(j)}(X^{o(j)}|X^{o(k)}, k \leq o(j))) = \sum_{o(j)} E(X^{o(j)}|X^{o(k)}, k \leq o(j))$$

The reordering $o_a$ is based on the attribute selectivity $s_{att} : A \rightarrow \mathbb{R}$, it follows the descending selectivity such that

$$\forall a_i, a_j \in A : s_{att}(a_i) \leq s_{att}(a_j) \Rightarrow o_a(i) > o_a(j)$$

We propose three measures for the attribute selectivity: The first considers only the attribute domains and profile distribution, the second also takes the event distribution into account, the third measure depends on the conditional distributions of the profiles – the shape of the tree.

**A1.** For each attribute, the ratio of the size of the zero-subdomain regarding the profile distribution and the domain-size: $s_{att}(a_j) = \frac{d_0(a_j)}{d_j}$.

**A2.** The ratio of the probability of the zero-subdomain and the probability of the domain-size under consideration of the profile distribution: $s_{att}(a_j) = \frac{d_0(a_j) * P_e(D_0(a_j))}{d_j}$. $P_e(D_0)$ denotes the probability that the data of an event take attribute values of $D_0$: $P_e(D_0) = P_e(X = x_0)$.

**A3.** The relative size of the zero-subdomains $D_0$ depending on the conditional probabilities. These probabilities influence the shape of the tree. The attributes have to be ordered in the tree such that the sum of the zero-subdomains is maximal. Up to $n! * 2(p-1)$ tree variations have to be evaluated to compute this measure.

The examples show the influence of the attribute-based measures $A1$ and $A2$:

***Example 7.6 (Enhanced Tree-based Filtering using Attribute-based Reordering)***
*Derived from the sizes of the attribute domains defined in Example 7.4, the selectivities of the attributes based on Measure A1 are:*

$$\begin{aligned} a_1: &\quad d_1 = 80 \quad d_0 = 50 \quad s_{att}(a_1) = 0.625 \\ a_2: &\quad d_2 = 100 \quad d_0 = 75 \quad s_{att}(a_2) = 0.75 \\ a_3: &\quad d_3 = 100 \quad d_0 = 0 \quad s_{att}(a_3) = 0 \end{aligned}$$

*We assume distribution $P_e(X^1)$ as in Example 7.5, and*

$$P_e(X^2) = ([0-30]:5\%, [30-80]:60\%, [80-90]:25\%, [90-100]:10\%), \text{ and}$$

$$P_e(X^3) = ([0-35]:90\%, [35-40]:5\%, [40-50]:2\%, [50-100]:3\%).$$

*For ease of computation, we assume independent attributes. The expectation for the original tree (Figure 7.10) is*

$$\begin{aligned} E(\sum_{j=1}^{3}(X^j|X^{j-1},..,X^1)) &= \\ E(X^1)+E(X^2|X^1)+E(X^3|X^2,X^1) &= 3.371 \end{aligned}$$

*Reordering according to Measure A1 leads to*

$$\begin{aligned} E(\sum_{o_a(j)=1}^{3}(X_a^o(j)|X^{o_a(j)-1},..)) &= \\ E(X^2)+E(X^1|X^2)+E(X^3|X^1,X^2) &= 1.91 \end{aligned}$$

*Reordering based on Measure A2 for the selectivities of the attributes leads to the same result:* $s_{att}(a_1) = 0.05$, $s_{att}(a_2) = 0.6$, $s_{att}(a_3) = 0$.

Finally, we apply both the value-based and the attribute-based reordering to our example:

***Example 7.7 (Enhanced Tree-based Filtering using Value- and Attribute-based Reordering)*** *The expectation for the example tree using the Measures V1 and A2 is* $E(\sum_{o_a(j)=1}^{3}(X^{o_a(j)}|X^{o_a(j)-1},..)) = 1.08$. *The resulting tree is depicted in Figure 7.11. Binary search in the attribute-reordered tree leads to* $E(\sum ..) = 1.616$.

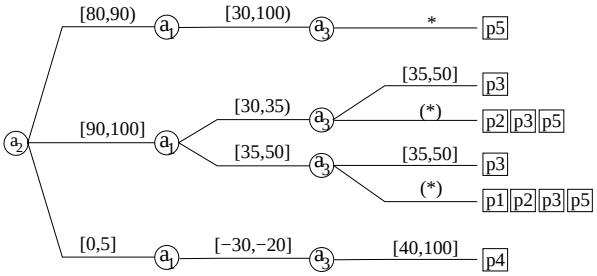



Figure 7.11: Reordered profile tree (Example 7.7)

The enhanced tree algorithm based on the proposed reordering of the profile tree is shown in Algorithm 7.2. The differences to the basic Algorithm 7.1 are emphasized.

In simulations, a performance gain of 90% has been recorded for selected scenarios that are typical for event notification services. We simulated the algorithm using the various measures proposed here, the measures $A1$ and $A2$ have been implemented in our A-MEDIAS system. The simulation and performance results are discussed in detail in Chapter 10.

**Algorithm 7.2** Enhanced tree-based algorithm: efficient tree-search for single matching path

**input**: profile tree with *reordered attribute levels* $a_i$ and *reordered branches* $b_j$
event-message with (attribute;value) pairs
**output**: list of matching profiles or NULL

1: current-node := root /* root = $a_j$ with $o_a(j) = 1$ */
2: **while** (current-node <> leaf-node) **do**
3: branch-to-follow := NULL
4: iterate sequentially through branches $o_v(i) \in [1, m]$
5: **if** branch predicate at $b_{o_v(i)}$ true for event-message
6: **then** branch-to-follow := $b_{o_v(i)}$
7: **if** (branch-to-follow <> NULL)
8 **then** follow the branch to the next node: current-node := branch-to-follow $\rightarrow a_{o_a(j)+1}$
9: **else** EXIT
10: **output** current-node

## 7.3 Composite-Event Filtering in a Single Step

After having studied efficient filtering of primitive events, this section concentrates on the filtering of composite events. The performance is measured as filter time: We define the filter time for a composite event as the time between the occurrence of the last event contributing to a composite event and the client notification. To the best of our knowledge, all existing methods for composite event detection consist of two steps: In the first step, the primitive profiles are evaluated and in the second step, the corresponding composite profiles are identified. Thus, the composite event is detected in a separate step after the filtering of primitive events. As we shall see, these two-step methods contain unnecessary filter operations. In this section, we propose a new method for the filtering of composite events that integrates the detection of composite events into the detection of primitive events: After the filtering of a primitive event, its contribution to a composite event is tested. In that way, the composite event is detected successively. No additional step is required for the identification of the composite event after the last contributing primitive event has been detected. The identification of the composite event is accelerated and the overall filtering time is reduced.

First, we analyze typical methods for the filtering of composite events and discuss their performance. Then, our approach of combined filtering of primitive and composite events is introduced.

### 7.3.1 Existing Methods for Composite-Event Filtering

We analyze existing methods for the filtering of composite events and point out their performance weaknesses. Our approach is introduced in the next section. We distinguish approaches using finite state automata (e.g., Ode [GJ91]), Petri Nets (in Samos [GD92]), or trees (e.g., READY [GKP99], GEM [MS97]). Here, we do not consider the specific conditions of distributed event filtering. For illustration, we use our example from the facility management scenario (see Scenario 1 on Page 12):

***Example 7.8 (Two-Step Filtering)***
*A building's technicians and service personnel are often interested in certain combinations of events, e.g., 'Notify if at an office first a window is broken and then the presence detector sends a signal' (cf. P3 in Table 5.1, Page 59). Using the notation introduced in Chapter 5 (while omitting the parameters),*

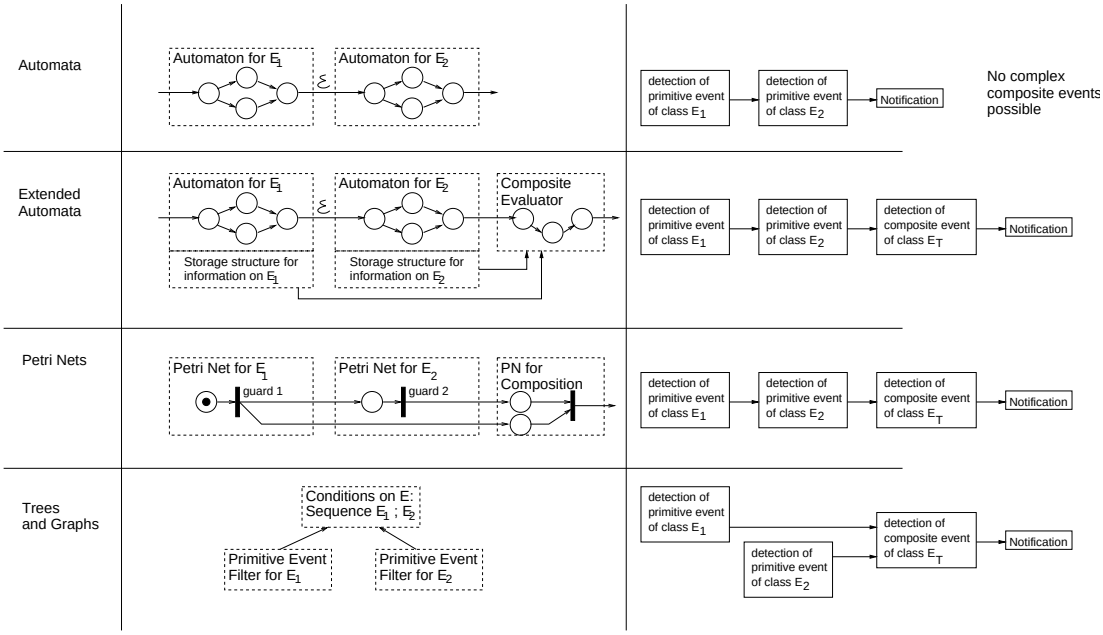



Figure 7.12: Existing methods for detection of composite events

*this profile defines the following event class* $E_T$

$$E_T = (E_1; E_2)_T$$

*where* $E_1$ *is the class of events regarding broken windows,* $E_2$ *the class of events regarding presence signals. The time span* $T$ *may be set to a particular epsilon, e.g., 5 minutes. Additionally, a profile may contain predicates regarding the composite event attributes, such as the predicate* $E_1.room = E_2.room$*. These additional conditions are called* binding predicates*. Figure 7.12 displays simplified structures for detecting the composite event instance of* $E_T$ *using the different methods described in this section.*

**Finite State Automata:** Composite event expressions are similar to regular expressions if they are not parameterized. Using this observation, it is possible to implement event expressions using finite automata. The first approach for using automata has been made in the active database system Ode [GJS92a]. An automaton's input are the primitive event components from the corresponding composite event as they occur in the history (see Figure 7.12, first row). If the automaton enters an accepting state after the input of a primitive event, then the composite event implemented by the automaton is said to occur at the time of this primitive event. Automata are not sufficient if binding predicates have to be supported. The automata have to be extended with a data structure for storing the additional event information of the primitive events from the time of their occurrence to the time at which the composite event is detected. This extension is shown in the second row in Figure 7.12.

**Petri Nets:** Petri Nets are used in several event-based systems to support the detection of complex composite events, for example in the active database system SAMOS [GD92], and the monitoring system HiFi [SWM99]. In a Petri Net created for a profile regarding a composite event, the input places refer

to primitive events, and the output places model the composite event. Each new profile describing a composite event causes the creation of the appropriate Petri Net. The incremental detection of composite events is described by the position of the markings in the Petri Net. The firing of the transition depends on the input tokens and the positive evaluation of the transition guards. The occurrence of the composite event is signaled as soon as the last element of a given sequence order is marked. Petri Net based composite event detection is shown in the third row in Figure 7.12.

**Matching Trees:** Another approach to implement composite-event filters uses matching trees that are constructed from profiles describing composite event structures. This method has been used in READY [GKP99] and Yeast [KR95]. The primitive event parts are the leaves of the matching tree, composites are the parent nodes in the tree hierarchy, as shown in the fourth row in Figure 7.12. Parent nodes are responsible for maintaining information for matched events, such as mapping of event variables and successfully matching event instances. This information is updated by child nodes on every match and is passed to the parent nodes. Parent nodes perform further filtering. A composite event is detected if the root node is reached and the respective event data are successfully filtered in the root node. Then, context-related information and additional information (e.g., client information) is passed to the notification component. Note that in tree-based composite filtering, components of a composite event may be filtered unnecessarily, e.g., the second event of a sequence is filtered even though the first event of the sequence did not occur.

**Graphs:** An approach very similar to the tree-based one described above is the graph-based detection of composite events. Here, each composite event is represented by a directed acyclic graph (DAG), where nodes are event descriptions and edges represent event composition, see also the fourth row in Figure 7.12. Nodes are marked with references to respective event occurrences. After event detection, parent nodes are informed and checked for consumption recursively. References to events are stored until consumption is possible. In addition to event composition edges, nodes are accompanied by rule objects that are fired after the corresponding event occurred. Graph-based composite detection is implemented in Sentinel [CM94] and Eve [GT96].

The drawbacks of the existing filter methods are illustrated in the following example. Here, we extend our Example 7.8:

***Example 7.9 (Performance of Two-Step Filtering)***
*For the profile $E_T = (E_1; E_2)_T$, we consider an example trace: $tr = \langle e_1, e_2, e_3, e_4, e_5, e_6, e_7 \rangle$ with $e_3, e_4, e_6 \in E_1$ and $e_1, e_2, e_5, e_7 \in E_2$. Figure 7.13 shows the event filtering using the different methods described above. We do not show the filtering of each incoming event but only depict the filter efforts contributing to the composite event. The events $e_3$ and $e_4$ do not form a composition with $e_5$, because the temporal distance is larger than $T$. Only the event instances $e_6$ and $e_7$ match our composite profile.*

*Composite-event filtering based on trees or graphs considers all matching primitive events, regardless of their order or temporal distance. This results in a large number of unnecessary filter operations. Composite-event filtering based on Petri Nets or extended automata considers only matching primitive*

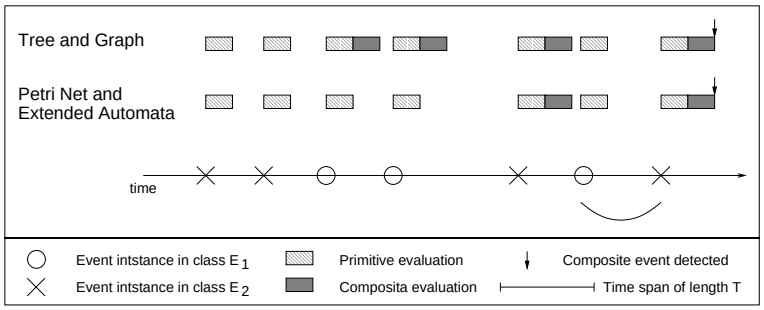



Figure 7.13: Composite event detection for example trace as described in Example 7.9

*events that are in the desired order. Matching of temporal conditions is performed during the composition phase. Unnecessary filter operations are performed for non-matching composites.*

All approaches for the filtering of composite events that are applicable to parameterized composite events (i.e., all but simple automata) have in common that two steps are necessary to identify composite events: The detection of primitive events followed by the evaluation of the composite binding predicates. Thus, the filter time is extended unnecessarily.

In the next section, we propose our method to identify the composite event within the single step of detecting the contributing primitive events. Our method provides increased performance for the filtering of primitive and composite events.

### 7.3.2 Composite Event Detection in a Single Step

We use the idea of partial evaluation [JGS93]: Primitive profiles contributing to composite ones are evaluated only if they potentially contribute to a composite event. For example, for the sequence of events $(E_1; E_2)$, first only the $E_1$-profile is evaluated. The $E_2$-profile is included into the evaluation process only after an event $e_1 \in E_1$ did occur.

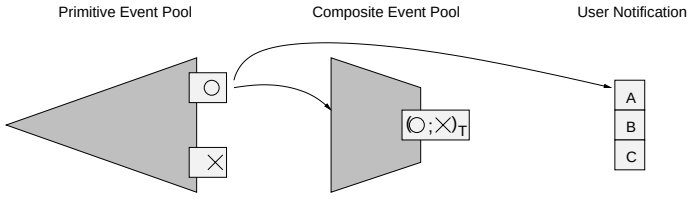
We illustrate the idea of composite filtering in a single step by pointing out the differences to the two-step filtering. Therefore, the process of filtering using a two-step algorithm is described in detail before our single-step algorithm is introduced. For illustration, we consider the following three example profiles:

Client A: $E_A = E_1$ (profile regarding primitive events)

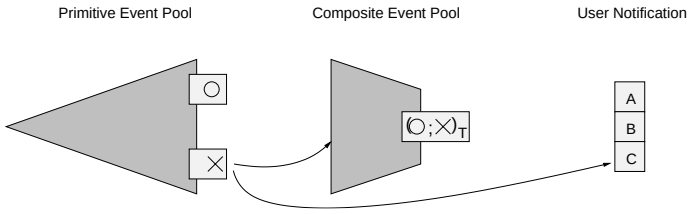
Client B: $E_B = (E_1; E_2)_T$ (profile regarding composite events as in Example 7.8)

Client C: $E_C = E_2$ (profile regarding primitive events)

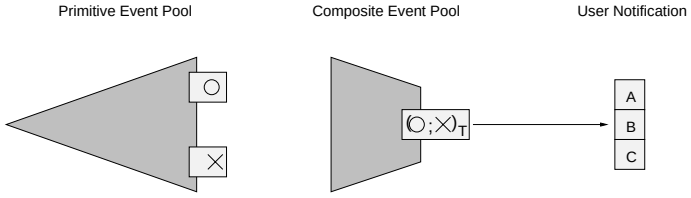
Figure 7.14 shows the principle of *composite event detection in two steps* for these three profiles: The triangle represents the primitive profile pool. The primitive profile pool represents a structure for indexing and filtering primitive profiles, e.g., using our enhanced primitive tree-based algorithm. In the two-step method, the pool contains *all* profiles regarding primitive events. Each incoming primitive event has to be filtered against all profiles in that pool. Clients with profiles regarding primitive events (i.e., clients A and C) are notified after the detection of these events. This detection of the primitive events is the first step in the event detection mechanism.




(a) Situation after occurrence of event $\circ \in E_1$




(b) Situation after occurrence of event $\times \in E_2$




(c) Composite-event filtering

Figure 7.14: Composite event detection using two-step methods, $\circ \hat{=} e_1 \in E_1$ and $\times \hat{=} e_2 \in E_2$

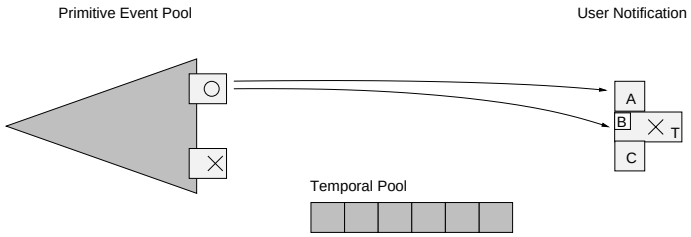
The results of the primitive filtering serve as input for the composite filtering (see figures 7.14(a) and 7.14(b)). The profiles regarding composites are stored in the composite pool, represented by the square in Figure 7.14. The incoming primitive events are assigned to the composite profiles. If all contributing events for a certain composite did occur, the composite event is signaled to the interested clients (client B in Figure 7.14(c)). If the time span between the primitive events is larger than $T$, the composite profile is not matched, and the detected primitive events are dismissed.

We now introduce our *composite event detection in a single step*. Figure 7.15 shows the principle of our algorithm: The primitive event pool and a temporal pool are required, the composite pool is not used. After the detection of a primitive event, the interested clients are notified (client A in Figure 7.15(a)).

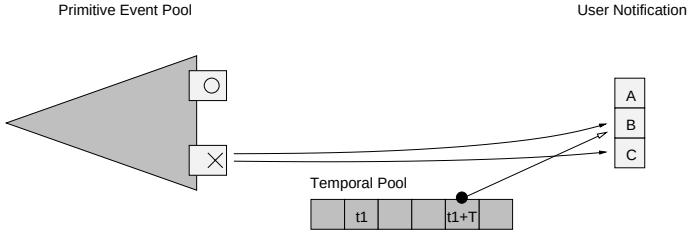
For storing the information about composite profiles, auxiliary profiles are created. After the match of a contributing primitive event, an internal notification regarding the auxiliary profile is created. This notification triggers the insertion of the remaining composite part into the primitive profile pool.[7]

Consider the auxiliary profile for the composite profile of client $B$: An internal auxiliary client is notified about the occurrence of the partial composite event (see Figure 7.15(a)). The auxiliary profiles for the internal client carry the information about the composite profile as well as the information about its partial evaluation.

[7]The remaining parts of profiles can be identified, for example, by analyzing the respective Petri Net for the composite profile.




(a) Situation after occurrence of event $\circ \in E_1$ at time $t1$




(b) Situation after occurrence of event $\times \in E_2$

Figure 7.15: Composite event detection using our single-step method, , $\circ \hat{=} e_1 \in E_1$ and $\times \hat{=} e_2 \in E_2$

In Figure 7.15(b), after the detection of event $\circ$, the profile for event $\times$ has been inserted into the pool. For the observation of the maximal time span $T$, an *auxiliary terminator reference* is inserted into the temporal pool. Terminator references cause the removal of the referenced profile from the pool. The temporal terminator in Figure 7.15(b) causes the removal of the profile for client B at time $t1+T$, where $t1$ is the time of the primitive event $\circ$.

After the match of the final contributing primitive event within the composite profile, a notification is sent to the client. Using our single-step method, the notification about the composite does not suffer additional delays due to additional filtering of binding predicates for the composition. Algorithm 7.3 shows the pseudocode for the sequence detection using our single-step method.[8]

**Algorithm 7.3** Single-Step Detection of Sequence $(E_1; E_2)_T$ (all duplicates, all pairs)

**input**: $e$ – event-message to be filtered
$P$ – profile pool with currently observed profiles
**output**: notification about $(E_1; E_2)_T$

1: initialize P with $E_1$, $(E_1 \Rightarrow E_2)$, and $(E_1 \Rightarrow T)$
2: **on** event e
3: **if** $(e \in E_1)$ **then**
4: **if** $(E_2 \notin P)$ **then** insert $E_2$ in P
5: set reference $r_1 : (E_2 \Rightarrow e)$
6: insert $t_1 : (t(e) + T)$ in P with reference $r2 : (t_1 \Rightarrow E_2)$
7: **if** $((e \in E_2)$ && $(\exists$ reference $r$ with $(E_2 \Rightarrow e)))$ **then** notify about $(E_1; E_2)_T$
8: **if** $(e =$ time event $t_1$ **)** **then** remove reference $r_1$
9: **if** (no references from $E_2$) **then** remove $E_2$ from P
10: remove $t_1$ from P

[8]We are aware of the fact that the efficiency of our single-step algorithm depends on sufficiently efficient algorithms for tree-maintenance. We discuss this aspect in the analysis of our performance tests in Chapter 10.

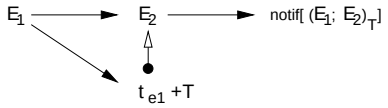
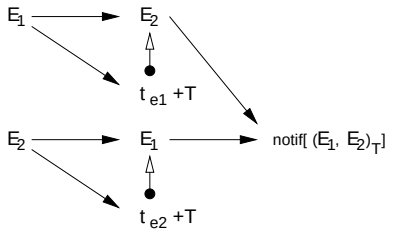
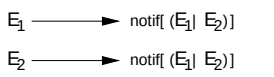
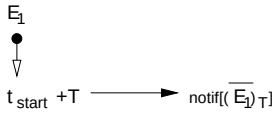
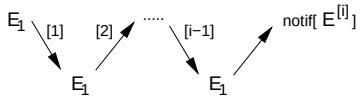
| Operator | Filter Procedure |
|---|---|
| Temporal Sequence<br>e.g., $(E_1; E_2)_T$ |  <br>insert profile $E_1$, after $e_1 \in E_1$ insert profile $E_2$ and time profile $t_{e_1} + T$ with terminator to profile $E_2$ |
| Temporal Conjunction<br>e.g., $(E_1, E_2)_T$ |  <br>insert profiles $E_1$ and $E_2$, after $e_1 \in E_1$ insert profile $E_2$ and time profile $t_{e_1} + T$ with terminator to profile $E_2$, react on $e_2 \in E_2$ respectively |
| Temporal Disjunction<br>e.g., $(E_1 \vert E_2)$ |  <br>insert profiles $E_1$ and $E_2$, notify after $e_1 \in E_1$ or $e_2 \in E_2$ |
| Negation<br>e.g., $(\overline{E_1})_T$ |  <br>insert profile $E_1$ with a terminator to the time profile $t_{start} + T$, notify after matching time profile, each $e_1 \in E_1$ starts the process anew |
| Selection<br>e.g., $(E_1^{[i]})$ |  <br>insert profile $E_1$, after $e_1 \in E_1$ insert profile $E_1$, notify after the $i^{th} e_1 \in E_1$ |
| Legend: A ⟶ B: after event A insert profile B  A •⟶▷ B: after event A remove profile B | |

Table 7.3: Profile handling for various composite operators using our single-step algorithm

Table 7.3 shows a graphical representation of the algorithm's principle. In analogy to the example for the two-step approach (see Example 7.9), we discuss the composite evaluation for an example trace in the following example.

***Example 7.10 (Performance of Single-Step Filtering)***

*Let us consider the profile $E_T = (E_1; E_2)_T$ and the example trace $tr = \langle e_1, e_2, e_3, e_4, e_5, e_6, e_7 \rangle$ with $e_1, e_2, e_5, e_7 \in E_1$ and $e_3, e_4, e_6 \in E_2$. Figure 7.16 shows our single-step filtering for that trace. Again, we only depict the filter efforts contributing to the composite event. Only those primitive events are evaluated that contribute to the composite event. Additionally, the composite event is detected earlier. Few unnecessary filter operations are performed for non-matching composites.*

*As we can see in Figure 7.16, the filtering costs of our single-step method are lower than the costs of the two-step methods (cf. Figure 7.13).*

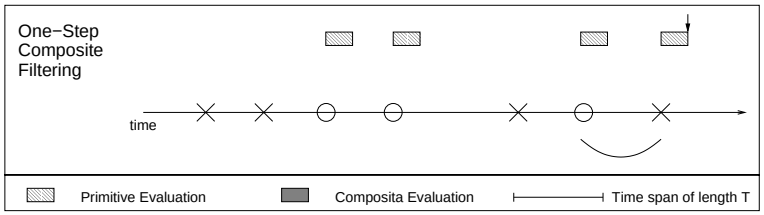



Figure 7.16: Composite event detection for example trace as described in Example 7.10

**Temporal Operators for Composite Events.** In Algorithm 7.3, we have shown the single-step filtering for a simple temporal sequence. The principle can be translated to the other temporal composition operators conjunction, disjunction, selection, and negation (see Table 7.3). In the table, for each temporal operator, we give the filter procedure that implements the respective single-step method. The filter procedure is defined both graphically and by description. In the graphics, we show the order of insertions into the profile and temporal pool. For example, the temporal disjunction $(E_1|E_2)$ requires the insertion of both the profiles regarding $E_1$ and $E_2$. A notification is sent after the detection of event instances either in $E_1$ or in $E_2$. For the negation $(\overline{E_1})_T$, terminator references are required: At starting time $t_{start}$ of the profile, a temporal profile regarding $E_1$ is inserted that triggers a notification after the time span $T$ at $t_{start} + T$. In the table, this is depicted by the arrow towards the notification. If an event instance of $E_1$ occurs within that time span, the temporal profile is removed and no notification is sent. This is depicted in the table by the terminator arrow from $E_1$. Different from the sequence handling, here the auxiliary terminator is attached to the primitive profile for the event in question. The temporal restriction provides the accepting reference to the client to be notified. As one example, the single-step algorithm for the negation (unique pair) is shown in Algorithm 7.4.

**Algorithm 7.4** Single-Step Detection of Negation $(\overline{E_1})_T$ unique occurrence

**input**: $e$ – event-message to be filtered
$P$ – profile pool with currently observed profiles
$t_{now}$ – refers to the current point in time
**output**: notification about $(\overline{E_1})_T$

1: initialize P with $E_1$, $t_1 : (t_{now} + T)$, and reference $r_1 : (t_1 \Rightarrow E_1)$
2: **on** event e
3: **if** ($e \in E_1$) **then**
4: remove reference $r_1$
5: **if** (no references from $t_1$) **then** remove $t_1$ from P
6: insert $t_1 : (t_{now} + T)$ in P with reference $r_1 : (t_1 \Rightarrow E_1)$
7: **if** ($e =$ time event $t_1$ ) **then**
8: notify about $(\overline{E_1})_T$
9: remove reference $r_1$ from P
10: **if** (no references from $t_1$) **then** remove $t_1$ from P

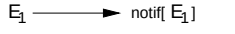
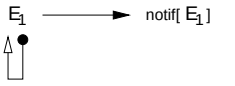
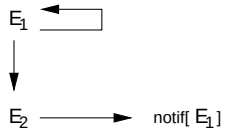
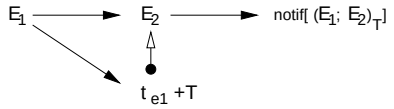
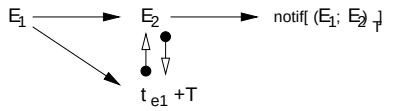
| Parameter | Filter Procedure |
|---|---|
| All Duplicates<br>e.g., regarding $E_1$ |  <br>do not remove profiles after match |
| First Duplicate<br>e.g., first $e_1 \in E_1$ |  <br>remove profile after match |
| Last Duplicate<br>e.g., first $e_1 \in E_1$<br>before $e_2 \in E_2$ |  <br>keep updating event information until last event instance |
| All Pairs<br>e.g., of $(E_1; E_2)_T$ |  <br>do not remove profiles after match |
| Unique Pairs<br>e.g., of $(E_1; E_2)_T$ |  <br>remove composite event information after match |
| Legend: A ⟶ B: after event A insert profile B  A ●⟶▷ B: after event A remove profile B | |

Table 7.4: Profile handling for various composite parameters using our single-step algorithm

**Parameterized Composite Events.** Profiles regarding composite events may also carry additional parameters, as introduced in Chapter 5. These parameters can also be supported in our single-step approach (see Table 7.4). The parameter for event instance detection influences which of the duplicated contributing events is stored. For example, for the *first* event in the duplicate list, the auxiliary profile refers to the remainder of the composite profile. For the *last* event, two auxiliary profiles have to be created: One of these profiles refers to the remainder of the composite profile (in case there is only a single event instance), and the other profile refers, again, to the already matched partial profile (to detect the duplicates).

The parameter for event consumption influences how long partially matching events have to be stored. As discussed in Chapter 5, we distinguish three modes: 'remove after match', 'hold after match', and 'reapply filter after remove'. For example, for 'remove after match' mode, the partially matching events are deleted after the composite has been found (i.e., only *unique event pairs* are detected). For the 'hold after match' mode, the auxiliary profiles for the remainder of the composite profile remain in the system (i.e., *all event instance combinations* are detected).

Selected algorithms of our single-step method have been implemented and tested in our A-MEDIAS system. Depending on the event and profile distribution, a performance gain of up to 85% has been recorded. The efficiency test results are discussed in detail in Chapter 10.

## 7.4 Related Work

Several filter algorithms have been proposed for event notification services, most of them are main memory algorithms (as is the algorithm proposed here). The algorithms that are directly related to our approaches for primitive and composite filtering have been discussed in the respective sections.

Our distribution-based filter method for primitive events is inspired by tree-based indexing strategies for keyword-based search, such as the ranked tree method [YGM94b]. In contrast to this unstructured search, our approach considers structured event messages and profiles and is therefore more expressive. For the tree-based algorithm used in Gryphon [SBC98, ASS+99], several optimizations have been proposed that are based on static tree-analysis. The optimizations mainly focus on don't-care edges and may be applied additionally to our tree reordering.

Closely related to our attribute-based reordering is the quenching algorithm in the Elvin system [SA97], which discards unnecessary information without consuming resources. In Siena [CRW01], the concept of early rejection on event level is used for a distributed service that implements profile and event propagation within a network. In contrast to our approach, the algorithms in Elvin and Siena prevent the filtering on event-message level and not on the content-level within the filtering. These orthogonal techniques may be used additionally. None of these algorithms use the profile and event distribution to achieve better performance.

Our single-step method for the filtering of composite events is inspired by ideas from partial evaluation [JGS93]. Applications for partial evaluation are programs with many similar subcomputations, highly parameterized computations, and database query optimization algorithms. Here, we do not strictly follow the rules of partial evaluation, namely substitution of values for program variables and program rewriting, but follow the basic idea of computing only as much as needed for the next program step.

Other filter approaches focus on the particular requirements of distributed services. A filtering based on distributed trees has been implemented in EPS [ME01]. Here, trees with identical subtrees may use common structures. In Eve [GT96] and GEM [MS97], identical subtrees are duplicated and may be traversed several times during filtering. Another approach to ensure better performance for distributed event filtering systems employs structure-based optimization using filter similarities [MFB02]. For example, a covering-function prevents the forwarding of profiles that are contained in profiles already known to a server. Methods for event composition in a distributed environment can be used together with our single-step approach. In our A-MEDIAS system, we integrate a selection of these methods with our filter algorithms.

## 7.5 Summary

We identified the performance of an ENS as a basic Requirement R6 (cf. Chapter 2). To analyze the influences on the system's performance, we introduced a performance model. The analysis of the model identified the filter as the most influential part for the system's runtime. We additionally emphasized the impact of event detection methods on the overall performance of the system. In this chapter, we proposed two algorithms for the efficient filtering of primitive and composite events.

For the filtering of primitive events, we introduced a distribution-dependent improvement of the fastest filtering algorithm [GS95] for event notification services. This algorithm is based on our analysis of a separate model for tree-based filters. Value-dependent and attribute-dependent selectivity measures have been introduced as a basis for the improved algorithm. We have shown analytically that our distribution-based approach improves the average-case performance of tree-based algorithms. Our approach is suitable to reduce the workload in resource critical environments, for example in mobile computing. Unnecessary event information is rejected as early as possible and the filtering can be optimized according to the application and client needs.

For the filtering of composite events, we presented a single-step algorithm. Current implementations use two-step algorithms that perform unnecessary filter operations. Our algorithm uses of the idea of partial evaluation: Only those profiles are evaluated that may directly contribute to a composite profile. No avoidable filter operations are performed.[9] After a general introduction of our single-step algorithm, we presented methods for the handling of profiles with various composition operators. Additionally, we discussed profile handling under various composition parameters.

Adaptation to changing event and profile distributions requires information about the distributions. How to determine this information dynamically in a running system is discussed in the next Chapter 8. Both algorithms are implemented in our A-MEDIAS system, the algorithm-specific implementation details are discussed in Chapter 9. Experimental efficiency results are presented in Chapter 10: For both algorithms introduced here, performance gains up to 85 to 90 percent have been recorded.

[9] However, unnecessary operations may be performed if only a single contributing event of a pair occurs. These operations cannot be avoided.