Efficient Filtering in Publish-Subscribe Systems using Binary Decision Diagrams

Alexis Campailla
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
USA
+1 (425) 706-1843
alexisca@microsoft.com

Sagar Chaki **Edmund Clarke**
Carnegie Mellon University
School of Computer Science
5000 Forbes Avenue
Pittsburgh, PA 15213
+1 (412) 268 (8102|2628)
(chaki|emc)@cs.cmu.edu

Somesh Jha
Computer Science Department
University of Wisconsin
Madison,WI 53706
USA
+1 (608) 262 9519
jha@cs.wisc.edu

Helmut Veith
Vienna University of Technology
Database and AI Group
Favoritenstr. 9
A-1040 Vienna, Austria
+43 (1) 58801 18431
veith@dbai.tuwien.ac.at

ABSTRACT

Implicit invocation or publish-subscribe has become an important architectural style for large-scale system design and evolution. The publish-subscribe style facilitates developing large-scale systems by composing separately developed components because the style permits loose coupling between various components. One of the major bottlenecks in using publish-subscribe systems for very large scale systems is the efficiency of filtering incoming messages, i.e., matching of published events with event subscriptions. This is a very challenging problem because in a realistic publish-subscribe system the number of subscriptions can be large. In this paper we present an approach for matching published events with subscriptions which scales to a large number of subscriptions. Our approach uses Binary Decision Diagrams, a compact data structure for representing boolean functions which has been successfully used in verification techniques such as model checking. Experimental results clearly demonstrate the efficiency of our approach.

Keywords

Publish/Subscribe Systems, Binary Decision Diagrams

1 INTRODUCTION

One of the major goals in software engineering is to develop architectural styles that facilitate composing separately designed and developed components into a large-scale system. Several architectural styles have been developed to achieve this goal. Some familiar styles include those based on remote procedure calls, shared variables, and asynchronous message passing. One important architectural style for system composition is implicit invocation [15, 27] or publish-subscribe (hereafter referred to as pub-sub). In the pub-sub style components interact via publishing messages and subscribing to classes of messages.

In a pub-sub system there are two types of components or clients publishers and subscribers, that exchange messages through a server that we call a *broker*. Components and clients, as well as events and messages will be used synonymously throughout the paper. In the most common scenario, publishers and subscribers have no reciprocal knowledge about each other. Therefore, one of the major advantages of the pub-sub style is that the various clients are loosely coupled. Publishers connect to the broker to publish events they want to make the world aware of, and subscribers connect to the broker to establish subscriptions, in which they specify the set of messages they are interested in receiving. The job of the broker is to match published messages with subscriptions and deliver to the subscribers the messages of interest. The subscriber's selection criteria are evaluated against the incoming messages by the broker on behalf of the subscribers. The algorithm for matching incoming messages with selection criteria of the subscribers is called the *filtering algorithm*. The component that implements the filtering algorithm is called the *filtering engine*. Pub-sub brokers can be seen as the combination of efficient and reliable multicast delivery with advanced filtering capabilities.

Applications of pub-sub systems are numerous. Systems for financial data and news dissemination, process monitoring, and distributed event notification, to name a few. Also, they are a core technology to enable event-driven distributed web applications. The pub-sub style has been used in programming environments [24] and operating systems, as well. The call back mechanism for signal handling in operating systems is a classic example of the pub-sub style. Mechanisms to support the pub-sub style are found in commercial toolkits (e.g, Softbench [16], ToolTalk [28], DecFuse), communications standards (e.g., Corba [12]), integration frameworks (e.g., JavaBeans [19]), and programming environments. Sun has recently announced a specification for a messaging service called the JAVA Message Service or JMS [22]. JMS also supports primitives to support the pub-sub style. Abstracting from the particular application, one can see that pub-sub systems have a very broad scope.

One of the major advantages of the pub-sub paradigm is that it allows filtering capabilities, i.e., a subscriber will only receive messages that satisfy a certain criteria. An event broadcast system, where all the information is broadcast to all consumers, makes poor use of network resources. In such a system filtering has to be done by the subscriber itself. This

Draft of paper submitted to ICSE 2001.

approach is feasible on a high-speed dedicated network, but becomes less attractive when the network resources are the primary bottlenecks. In this case, it is beneficial to perform filtering as close as possible to the information source. Having the source know more about the specific interests of the subscribers allows building systems to a scale that would otherwise be impossible.

The main focus of this paper is on the filtering algorithm. Recall that a filtering algorithm matches an incoming message against the selection criteria of the subscribers in order to determine the set of subscribers interested in the message. Our goal is to provide filtering algorithms that can be deployed in large scale pub-sub systems, for example, pub-sub systems used for content distribution on the web. First, we provide some examples of typical uses of pub-sub style. Based on these typical examples we derive functional requirements and design goals for a filtering engine.

Example 1 [Financial data]
The most advanced pub-sub systems of today are probably the ones used in electronic trading floors for delivery of financial data such as stock quotes, company news, and order confirmation events. These systems have historically been deployed in high-speed private networks, but are now beginning to appear massively on the Internet. These application have very high volumes of published messages. The number of subscriptions is moderate in intranet scenarios, but can be very high in Internet scenarios. For example, with many pub-sub applications, extensive filtering capabilities bring additional value. Many services are now offering features that let customers not only subscribe to a particular stock symbol, but also specify more advanced criteria, such as notification when the price hits a certain threshold.

Example 2 [News dissemination (e-commerce)]
Another very general application of pub-sub is news dissemination. E-magazines broadcast information updates regarding particular topics, and allow the users to select the topics they are interested in. One significant case that falls into this category is e-commerce oriented news dissemination. Most e-commerce sites let their users subscribe to news about products they are interested in. This scenario is usually comprised of a small number of publishers (mainly the magazine or e-commerce site), and a very high number of subscribers.

Functional requirements for a filtering engine

- *Matching published events with subscriptions*
  The filtering engine must be able to evaluate a set of subscriptions against an incoming stream of messages subject to the following requirements:

  • *Expressivity.* The language for expressing the subscription criteria must be rich. For example, a language for expressing subscription criteria [1] is described in detail in JMS [22].

  • *Efficiency.* The matching must be done extremely efficiently in real time.

  • *Scalability.* The matching must handle very large numbers of subscriptions.

- *Add and remove subscriptions*
  The filtering engine must be able to add and remove subscriptions from the set of existing subscriptions.

Design goals
Our main goal in this paper is to design a filtering engine that is *efficient and scalable*. In a realistic system the number of subscriptions will be very large, e.g. a few hundred thousand. When a message is published by a client, the interested subscribers have to be quickly notified. Typically, the arriving messages satisfy very few subscription criteria. Consider the financial data example provided earlier. Suppose a message about *IBM* arrives. Although the total number of subscribers might be very large, typically, a small fraction of the subscribers will be interested in a specific stock such as *IBM*. We call this the *irrelevance property*. We want to leverage the irrelevance property of pub-sub systems in the design of the filtering algorithm. Our goal is to match messages with subscriptions in real-time. We also want our matching algorithms to be scalable, e.g., to be able to handle few hundred thousand subscriptions.

The cost of adding, removing, and updating subscriptions is also addressed. However, the performance of these operations is not critical because they are not as frequent as the arrival of messages.

The strategies for distributing the filtering load of a pub-sub server across multiple hosts are not addressed in this paper. Even with strategies for load distribution in place, there is still a need for an efficient filtering engine that matches incoming messages against the subscription criteria of the subscribers.

2 OVERVIEW AND MAJOR CONTRIBUTIONS

Binary Decision Diagrams (or BDDs) are compact data structures for representing boolean functions [5]. BDDs have been successfully used in verification methods such as model checking [10]. We use BDDs to design a fast filtering algorithm. This section provides an overview of our approach.

First, each atomic formula in a client's subscription criteria or subscription (e.g. $company \in \{\text{IBM, Dell}\}$) is assigned a boolean variable. Notice that now each subscription is a boolean function of the boolean variables corresponding to the atomic formulae. Next each subscription is represented as a BDD. Intuitively, if there are several subscriptions in a pub-sub system, they share many common sub-expressions. For example, the following pattern might appear in several

[1] subscription criteria are called message selectors in the JMS parlance

subscriptions:

$$company \in \{IBM, Dell\} \text{ AND}$$
$$price \leq 1000$$

By representing subscriptions as BDDs we can exploit the commonality (i.e., shared sub-subscriptions) between different subscriptions. A published message is now simply a partial assignment to the boolean variables corresponding to the atomic formulas. When a message arrives, the set of subscriptions that "match" the message is found by a backward traversal algorithm on the corresponding BDDs. The efficiency of our method stems from the fact that common subexpressions, corresponding to the BDD nodes, are evaluated only once.

We believe that this paper makes two major contributions. First, we provide a precise semantics of when a message "matches" a subscription. This is not a trivial task because our language for describing subscriptions is rich and allows partial messages, i.e., messages that only refer to a subset of attribute variables. Second, we present an efficient and scalable filtering algorithm based on BDDs. Experimental results clearly demonstrate that our filtering algorithm is scalable.

3 RELATED WORK

Several applications of pub-sub systems were described earlier. *Content-based* pub-sub systems are intended for content distribution over a distributed network. In content-based pub-sub systems the subscription criteria filter messages based on their content. On the other hand, in *channel-based* pub-sub systems the subscription criteria filters messages based on the channels or ports they originate from. Since the benefits of our filtering algorithm only become apparent when the number of subscribers is very large (a typical case for content-based pub-sub systems), techniques presented in this paper are most suitable for content-based pub-sub systems. Notice that examples presented earlier fall into this category. To our knowledge the most important content-based pub-sub systems are GRYPHON [3, 1, 2, 18], SIENA [7, 9, 8], ELVIN [25, 14, 17] and KERYX [21, 20]. The subscription languages of the mentioned content-based pub-sub systems are fairly similar to each other and the subscription language presented in this paper. Our work focuses on the filtering algorithm suitable for content-based pub-sub systems. Techniques presented in this paper can be applied to improve the existing pub-sub systems. In the rest of this section, we compare our approach to the filtering engines of GRYPHON, SIENA, ELVIN and KERYX, and indicate how BDD-based filtering techniques can be incorporated into existing systems.

Among the four filtering engines, the filtering engine of GRYPHON is most similar to ours. It is based on parallel search trees, essentially decision trees labelled by atomic formulas. Optimization procedures are used to decrease the redundancy of the search trees. Although no details are provided, the authors mention that their optimizations may transform the search tree into an acyclic graph. Thus, it appears that a subset of BDD operations has been reinvented for parallel search trees. A disadvantage of GRYPHON is the restricted subscription language which consists only of conjunctions. The impressive complexity results of [1] (matching can be done in time sublinear in the number of subscriptions) are also natural properties of the BDD data structure. However, GRYPHON's restriction to conjunctions forces the user to express certain natural filters by submitting an exponential number of subscriptions. For example, the subscription

$$car \in \{GM, BMW, VW\} \text{ AND } name \in \{\text{Liz , Bob, Al}\}$$

$$\text{AND } (year < 78 \text{ OR } year = 80 \text{ OR } year > 90)$$

requires 27 GRYPHON-like subscriptions. In contrast, such *disjunctive* subscriptions are naturally handled by BDDs. Our experiments in section 8 demonstrate that our filtering engine is competitive in comparison to GRYPHON. We expect that the routing techniques developed for GRYPHON can be readily extended to BDDs.

In SIENA, the subscriptions are partially ordered with respect to subsumption[2]. This information is used to pre-filter events and forward them to other filtering engines accordingly. Although an elaborate automata-based matching algorithm akin to GRYPHON's has been described in [17], there is no implementation we are aware of. Similar to GRYPHON, SIENA has a conjunctive subscription language.However, in contrast to the other systems SIENA has a pattern concept which allows relating different messages to each other. The techniques developed in GRYPHON are complementary to the techniques of SIENA [7]. Our filtering engine, too, can be adapted to SIENA.

Very recently, a filtering algorithm for the system *Le Subscribe* has been presented [23] which uses indexing techniques for fast matching of atomic formulas, and clusters subscriptions to minimize cache failures. The clustering and matching methods of Le Subscribe however are restricted to conjunctive subscriptions similar as in Gryphon and Siena.

The subscription language of ELVIN is more expressive than the previous ones, and extends our subscription language by the ability to use regular expressions for string attributes. However, ELVIN does not have a filtering engine similar to ours or GRYPHON's. The BDD-based approach can be easily extended to handle more complicated atomic properties such as ELVIN's regular expressions. Therefore, the BDD filtering engine can be used in the context of ELVIN, as well.

KERYX is a Java notification service whose distributed architecture is similar to that of USENET. The subscription language DFL of KERYX is a LISP-like formalism which makes it hard to devise efficient filtering techniques. In future research we will investigate if BDD-based methods can be used for DFL or a fragment thereof.

[2]Subscription S subsumes subscription T if the set of messages described by S is a superset of the set of messages described by T.

As a general observation, a trade-off between expressive subscription languages and highly efficient filtering engines is characteristic of existing systems. Our work on BDD-based filtering engines helps to alleviate this problem.

4 THE SUBSCRIPTION QUERY LANGUAGE

Languages used to describe subscription criteria or subscriptions are called *the subscription query languages* or *query languages* for short[3]. In this section, we describe query languages SiSL, StSL, and DeSL which are used to submit subscriptions. In our framework query languages can be made more expressive. For example, we could use the language used in JMS for expressing subscriptions or message selectors in the JMS parlance [22]. To some extent, expressiveness of query languages is independent of our BDD-based implementation of the system.

Messages and Attributes. Each event is described by a set of *attributes*. We distinguish three types of attributes (integer, double, and string.) We fix a finite sequence $V = \langle v_1, \ldots, v_n \rangle$ of attributes (variables). Each attribute v_i has a type $\mathrm{type}(v_i) \in \{\mathrm{INT}, \mathrm{DBL}, \mathrm{STR}\}$, and a corresponding domain $D_{\mathrm{type}(v_i)}$ of constants. The tuple $\langle \mathrm{type}(v_1), \ldots, \mathrm{type}(v_n) \rangle$ is called the *event schema*.

In our current implementation, INT attributes are interpreted over 32 bit integers, DBL attributes are interpreted over 64 bit double precision floating point numbers, and STR attributes are ASCII strings of arbitrary length. It is easy to extend our methodology to a larger number of different attribute types.

A *message* simply assigns values to some (not necessarily all) of the attributes. Formally, a message is a partial assignment to the attributes, i.e., a mapping $m : V \to \bigcup_{v \in V} D_{\mathrm{type}(v)} \cup \{\star\}$ such that for each attribute v, either $m(v) \in D_{\mathrm{type}(v)}$, or $m(V) = \star$. If $m(v) = \star$, we say that m does not define v. A message m is *total* if it defines all attributes in V.

Example 3 [Sales Announcements]
Suppose that $V = \langle company, product, price \rangle$ *over event schema* $\langle \mathrm{STR}, \mathrm{STR}, \mathrm{DBL} \rangle$. *In practice, messages are likely to be given as simple XML-style documents. Consider the following message:*

```
<company> IBM </company>
<product> PC AT, 20 Mhz, 256 KB RAM
</product>
<price> 5000 </price>
```

This document describes a total message m_1 *where* $m_1(company) =$ *"IBM",* $m_1(product) =$ *"PC AT, 20 Mhz, 256 KB RAM", and* $m_1(price) = 5000$.

The following document however describes a different message m_2 *which is not total, because the attribute* $price$ *is not defined. Therefore,* $m_2(price) = \star$.

```
<company> Future Inc.  </company>
<product> 100 GHz Super PC </product>
```

Synopsis of Query Languages. We will now describe different subscription query languages of increasing complexity.

1. The *Simple Subscription Language* **SiSL** is used in settings, where all messages are total. SiSL provides a simple and powerful tool to select messages of a known format, typically in a non-distributed setting, or for specialized applications.

2. The *Strict Subscription Language* **StSL** requires that in order to match an StSL query, a message must define all attributes which occur in the query. We shall see later that the filtering engine presented in this paper is particularly efficient for the StSL case.

3. In the *Default Subscription Language* **DeSL** all attributes are initialized to default values, and updated by the message. Using the default value, it is also possible to test if individual attributes are defined by a message. Thus, DeSL extends the functionality of SiSL to heterogeneous message formats as it is often the case in distributed settings. A semantics, called the *NULL* semantics for queries is also provided by JMS [22]. Default subscription language can model the *NULL* semantics used in JMS.

SiSL: Simple Subscription Language for Total Messages. SiSL queries (subscriptions) are given by atomic properties of event attributes and Boolean combinations thereof.

Formally, SiSL is defined as follows: Let v be an attribute in V. If $\mathrm{type}(v) = \mathrm{INT}$, i.e., if v has type INT, and $c \in D_{\mathrm{INT}}$, then the formulas

$$v = c,\ v < c,\ c < v$$

are atomic formulas. If $\mathrm{type}(v) = \mathrm{DBL}$, atomic formulas are defined analogously. If $\mathrm{type}(v) = \mathrm{STR}$, and $s \in D_{\mathrm{STR}}$ is a string, then

$$v = s \text{ and } v \sqsubseteq s$$

are atomic formulas, where $v \sqsubseteq s$ denotes that v is a substring of s.

The set of atomic formulas is denoted by Atoms. A query $\mathcal{Q}$ is a Boolean combination of atomic formulas. $\mathrm{var}(\mathcal{Q})$ is the set of attributes occurring in $\mathcal{Q}$. $\mathrm{Atoms}(\mathcal{Q})$ is the set of atomic formulas occurring in $\mathcal{Q}$.

We use the following abbreviations in the query language:

- $x \leq y$ is an abbreviation for $x = y$ OR $x < y$.
- $x \in [a, b]$ is an abbreviation for $x \geq a$ AND $x \leq b$.
- $x \in \{a_1, \ldots, a_n\}$ is an abbreviation for $x = a_1$ OR $\ldots x = a_n$.

[3] Languages for expressing subscriptions resemble database query languages, hence the name.

Example 4 [Sales Announcements, ctd.]
The SiSL query

$$company \in \{"IBM", "Dell", "Siemens"\} \text{ AND } \\ "PC" \sqsubseteq product \text{ AND } "1000Mhz" \sqsubseteq product \text{ AND } \\ price \leq 1000$$

matches all announcements for 1000 Mhz PCs manufactured by IBM, Dell or Siemens which cost at most $1000.

The instantiation of a query $\mathcal{Q}$ by a message m is denoted $\mathcal{Q}^m$. Formally, we define $\mathcal{Q}^m$ as the query obtained from $\mathcal{Q}$ by replacing all variables $v \in \text{var}(\mathcal{Q})$ for which $m(v) \neq \star$ by $m(v)$.

Let $\mathcal{Q}$ be a SiSL query, and m be a total message. We say that $\mathcal{Q}$ matches m if $\mathcal{Q}^m$ evaluates to true. Formally, we write $m \models \mathcal{Q}$.

We will now describe extensions of SiSL which can handle non-total messages.

StSL: Strict Subscription Language. This language is the simplest extension of SiSL. A message m is *adequate* for a query $\mathcal{Q}$, if m defines all variables occurring in $\mathcal{Q}$, (i.e., if for all $v \in \text{var}(\mathcal{Q})$, it holds that $m(v) \neq \star$.) Note that total messages are adequate for arbitrary queries.

We define that m matches $\mathcal{Q}$ under strict semantics, iff m is adequate for $\mathcal{Q}$, and $m \models \mathcal{Q}$. We write $m \models_s \mathcal{Q}$ to denote matching under StSL semantics.

Remark: Note that the notion of adequateness is based on *syntactic properties* of the query. An alternative semantics may consider a query adequate if all attributes on which the query *depends* are defined. Let us call this notion *semantic adequacy*. With respect to semantic adequacy, queries like y AND $(x$ OR $\neg x)$ would be treated like y.We prefer the given syntactic notion because it is simpler for the user and detecting irrelevant variables is a hard problem and it avoids the paradox outlined in the following example.

Example 5 [Semantic Adequacy]
Suppose that Alice subscribes to messages where $price < 1000$ *and Bob subscribes to messages where* $price > 500$. *Later, they decide to join efforts, and subscribe using the disjunction* $price < 1000$ OR $price > 500$ *which is equivalent to true. According to semantic adequacy, Alice and Bob will now receive messages where no price is specified at all. Frustrated, Alice and Bob figure out that there is no way for them to receive all messages where* $price$ *is defined.*

The same kind of paradox will reappear on a technical level in connection with BDD restrictions (to be defined later.)

DeSL: Default Subscription Language. For each attribute v_i, we fix a *default value* $\mathbf{default}_{v_i} \in D_{\text{type}(v_i)}$. If a message m does not specify an attribute v, then $\mathbf{default}_v$ is used as default value. Formally, the *default extension* m_d of m is a total message defined as follows:

$$m_d(v) = \begin{cases} m(v) & \text{if } m(v) \neq \star \\ \mathbf{default}_v & \text{otherwise} \end{cases}$$

$\mathcal{Q}$ matches m under default semantics (in symbols, $m \models_d \mathcal{Q}$) if $\mathcal{Q}^{m_d}$ evaluates to true, i.e., if $m_d \models \mathcal{Q}$.

Example 6 *If* v *is a string attribute, it is often natural to define* $\mathbf{default}_v$ *to be the empty string. For integer and double values,* -1 *may be used to denote the absence of a valid attribute.*

As mentioned above, DeSL can be used to test if an attribute is defined at all in the following way: we extend each domain $D_{\text{type}(v)}$ by a new element $\perp$, and set $\mathbf{default}_v = \perp$, i.e. $\perp$ becomes the default value for all attributes. It is easy to see that the formula $v \neq \perp$ expresses that attribute v is defined. The following proposition is easy to prove.

Proposition 7 *Over total messages, SiSL, StSL, and DeSL are equally expressive.*

5 BINARY DECISION DIAGRAMS

In this section, we formally describe Ordered Binary Decisions Diagrams (BDDs), and the standard algorithms on BDDs.

BDDs. Let A be a set of propositional variables, and $\prec$ a linear order (also referred to as variable ordering) on A. An *ordered binary decision diagram* (BDD) $\mathcal{O}$ over A is an acyclic graph (V, E) whose non-terminal vertices (*nodes*) are labeled by variables from A, and whose edges and terminal nodes are labeled by $0, 1$. Each non-terminal node v has out-degree 2, such that one of its outgoing edges is labeled 0 (the *low edge* or *else-edge*), and the other is labeled 1 (the *high edge* or *then-edge*). If v has label a_i and the successors of v are labeled a_j, a_k, then $a_i \prec a_j$ and $a_i \prec a_k$. In other words, for each path, the sequence of labels along the path is strictly increasing with respect to $\prec$.

Each BDD node represents a Boolean function. The terminal nodes of $\mathcal{O}$ represent the constant functions given by their labels. A non-terminal node v with label a_i whose successors at the high and low edges are u and w respectively, defines the function $\mathcal{O}_v := (\neg a_i \text{ AND } \mathcal{O}_u) \text{ OR } (a_i \text{ AND } \mathcal{O}_w)$. We denote the Boolean function $(\neg a_i \text{ AND } x) \text{ OR } (a_i \text{ AND } y)$ by $\text{ITE}(a_i, x, y)$.

Example 8 *The BDD in Figure 1 represents the Boolean function* x AND $(y$ OR $z)$. *The variable ordering is* $x \prec y \prec z$.

The size of a BDD is the number of nodes of the BDD. It is well-known [5, 6] that for every variable ordering A and sequence $\langle f_1, \ldots, f_n \rangle$ of Boolean functions over A there exists a unique minimal BDD $\mathcal{O}$ over A with distinguished output nodes $o_1, \ldots, o_n$ which represent the Boolean functions $f_1, \ldots, f_n$. $\mathcal{O}$ is called the *shared BDD* for $f_1, \ldots, f_n$. The BDD $\mathcal{O}$ together with its output nodes $o_1, \ldots, o_n$ is denoted by $\text{BDD}_\prec(f_1, \ldots, f_n) = (\mathcal{O}, o_1, \ldots, o_n)$. $\text{BDD}_\prec(f_1, \ldots, f_n)$ is computable from

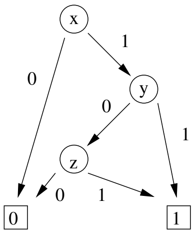

Figure 1: A BDD for function x AND $(y$ OR $z)$.

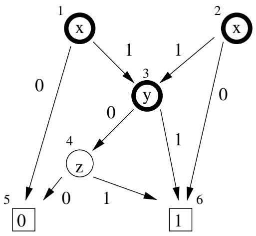

Figure 2: A shared BDD.

any other shared BDD over A for $\langle f_1, \ldots, f_n \rangle$ in polynomial time. The BDD $\mathrm{BDD}_{\prec}(f_1, \ldots, f_n)$ contains at most two non-terminal nodes, and no two nodes describe the same Boolean function.

Example 9 *Consider the shared BDD $(\mathcal{O}, 1, 2, 3)$ of Figure 2. In the figure, the BDD nodes are identified by natural numbers from 1 to 6, and output nodes are encircled by a thick line. The two root nodes 1 and 2 represent the functions x AND $(y$ OR $z)$ and $\neg x$ AND $(y$ OR $z)$ respectively. Moreover, node 3 represents the function $(y$ OR $z)$. Thus,* $\mathrm{BDD}_{\prec}(x$ AND $(y$ OR $z), \neg x$ AND $(y$ OR $z), (y$ OR $z)) = (\mathcal{O}, 1, 2, 3)$.

The size of $\mathrm{BDD}_{\prec}$ in general depends on the variable ordering $\prec$, and may be exponential in $|A|$. Effective algorithms for computing $\mathrm{BDD}_{\prec}$ and handling BDDs have been described in the literature [5]. In particular, operations on boolean functions (such as AND , OR , and $\neg$) can be efficiently performed using BDDs.

The main advantage of representing several Boolean functions by shared BDDs is given by the possibility of representing shared subfunctions only once. We shall see that this is important for the BDD filtering engine described in this paper. In the case of $n = 1$, v_1 is always the unique root of the BDD. Example 9 shows that an output node may have incoming edges. In the following, we shall always assume that BDDs are shared unless stated otherwise.

BDD data structure and evaluation.

We represent a BDD with n nodes by a graph whose vertices are the natural numbers $1, \ldots, n$. The adjacency relation is described by an array of size n such that the i-th element of the array is a record $(low[i], high[i], label[i], value[i])$, where $low[i]$ denotes the low successor of i, $high[i]$ denotes the high successor of i, and $label[i]$ denotes the label of node i. We will use $value[i]$ later to store results of the BDD evaluation.

Algorithm EvalBDD$(\mathcal{O}, \alpha)$
for $v := n$ downto 1
  if v is terminal
    then $value[v] := label[i]$
    else $value[v] :=$
      $\mathrm{ITE}(\alpha(label[v]), value[low[v]], value[high[v]])$
output $value$

Figure 3: BDD Evaluation Algorithm

We assume that the natural order of the nodes is consistent with the order of the variables, i.e., if i and j are non-terminal nodes such that $i < j$ then $label[i] \preceq label[j]$. Thus, the nodes are topologically sorted with respect to the variable order.

Let α be a truth assignment to A, i.e., a mapping $\alpha : A \to \{0, 1\}$. Given a Boolean function φ, $\varphi(\alpha)$ denotes the output of φ under assignment α. The algorithm **EvalBDD** in Figure 3 computes the value of each node in $\mathcal{O}$ under assignment α in $value$. We write $\mathbf{EvalBDD}(\mathcal{O}, \alpha)[i]$ to denote the component $value$ of the $i-$th element of the array.

Proposition 10 *Let $f_1, \ldots, f_n$ be a sequence of Boolean functions over $(A, \prec)$, and $\mathrm{BDD}_{\prec}(f_1, \ldots, f_n) = (\mathcal{O}, o_1, \ldots, o_n)$ its binary decision diagram. Then for all $1 \leq i \leq n$ and assignments α*

$$f_i(\alpha) \quad = \quad \mathbf{EvalBDD}(\mathcal{O}, \alpha)[o_i].$$

BDD restrictions. BDDs are intended to represent Boolean functions. However, it is possible that in certain situations, the BDD has to be correct only for certain inputs, because an external constraint enforces that no other inputs can occur. Let g be a Boolean function, and $\mathcal{O}$ be its BDD, and f be a Boolean function which evaluates to 1 on an input if the input is "relevant" to g. Then the BDD $\mathcal{O}$ can be replaced by a possibly smaller BDD $\mathcal{U}$ which is equivalent to $\mathcal{O}$ for all inputs satisfying f. Such a BDD $\mathcal{U}$ is called an $f-$restriction of $\mathcal{O}$. State-of-the-art BDD packages such as [26] employ heuristics to compute small restriction of BDDs. Note however that $\mathcal{U}$ in general represents a Boolean function *different from* f.

Formally, let $(\mathcal{O}, o_1, \ldots, o_n)$ and $(\mathcal{U}, u_1, \ldots, u_n)$ be BDDs over A, and let f denote a Boolean function over A. Then $\mathcal{O} \equiv_f \mathcal{U}$ if

$$f \to \left(\bigwedge_{1 \leq i \leq n} \mathcal{O}_{o_i} \equiv \mathcal{U}_{u_i} \right)$$

is tautologically true. $\mathcal{U}$ is called an $f-$restriction of $\mathcal{O}$, and vice versa. Intuitively, $\mathcal{O}$ and $\mathcal{U}$ are required to be equivalent for all assignments α for which $f(\alpha) = 1$. $\mathrm{Restrict}(\mathcal{O}, f)$

denotes the set of restrictions $\{\mathcal{U} : \mathcal{O} \equiv_f \mathcal{U}\}$ of BDDs which are $\equiv_f$ equivalent to $\mathcal{U}$. [4]

Remark: Note that $\mathcal{O}$ AND f is an $f-$restriction of $\mathcal{O}$. In this particular case, $\mathcal{O}$ is replaced by a BDD which evaluates to 0 for all inputs which do not satisfy f. $\mathcal{O}$ AND f does in general not yield a small $f-$restriction. Actually, $\mathcal{O}$ AND f may even be larger than $\mathcal{O}$.

6 THE FILTERING ENGINE

In this section, we describe the principles of the BDD filtering engine and the basic algorithms.

Query BDDs

The basic idea of query BDDs is to represent a large number of queries (subscriptions) by a shared BDD whose nodes correspond to atomic subformulas of the queries. Then, messages can be matched against queries by BDD evaluation algorithms.

Formally, let $\mathcal{Q} = \langle \mathcal{Q}_1, \ldots, \mathcal{Q}_n \rangle$ be a sequence of queries over the set of attributes V. Let $A = \bigcup_{1 \leq i \leq n} \mathrm{Atoms}(\mathcal{Q}_i)$ be the set of atomic subformulas of the queries. $\lceil A \rceil$ is a set of propositional variables, such that with each $a \in A$, we associate a propositional variable $\lceil a \rceil \in \lceil A \rceil$. Each query $\mathcal{Q}_i$ now corresponds to a propositional formula $\lceil \mathcal{Q}_i \rceil$ over $\lceil A \rceil$ which is obtained by replacing all occurrences of atomic subformulas a by their corresponding propositional variables $\lceil a \rceil$.

Given a linear order $\prec$ on $\lceil A \rceil$, $\mathrm{BDD}_\prec(\mathcal{Q})$ denotes the BDD for the Boolean formulas $\lceil \mathcal{Q}_1 \rceil, \ldots, \lceil \mathcal{Q}_n \rceil$, i.e., $\mathrm{BDD}_\prec(\lceil \mathcal{Q}_1 \rceil, \ldots, \lceil \mathcal{Q}_n \rceil)$ (we shall discuss later how to choose the variable ordering $\prec$). For each query $\mathcal{Q}_i$, $out(\mathcal{Q}_i)$ denotes the output node for $\lceil \mathcal{Q}_i \rceil$ in $\mathrm{BDD}_\prec(\mathcal{Q})$.

Example 11 *Suppose that Alice and Bob use the following two subscriptions:*

$\mathcal{Q}_1 := \quad price < 1500$ AND $"IBM" \sqsubseteq company$

$\mathcal{Q}_2 := \quad price < 1500$ AND $"McDonalds" \sqsubseteq company$

Here, the set $\lceil A \rceil$ of BDD variables is given by $\{\lceil price < 1500 \rceil, \lceil "IBM" \sqsubseteq company \rceil, \lceil "McDonalds" \sqsubseteq company \rceil\}$. We choose the variable ordering $\prec$ as follows: $\lceil "IBM" \sqsubseteq company \rceil \prec \lceil "McDonalds" \sqsubseteq company \rceil \prec \lceil price < 1500 \rceil$. The BDD $\mathrm{BDD}_\prec(\langle \mathcal{Q}_1, \mathcal{Q}_2 \rangle)$ is shown in Figure 4.

Each message m defines a partial assignment α_m to the BDD variables in the natural way, i.e.,

$$\alpha_m(\lceil a \rceil) = \begin{cases} \text{true} & \text{if } m \models a \\ \star & \text{if } m(\mathrm{var}(a)) = \star \\ & \text{i.e., if } m \text{ does not define the variable in } a \\ \text{false} & \text{otherwise} \end{cases}$$

If m is total, then $\alpha_m(\lceil a \rceil)$ is true if and only if $m \models a$.

[4]Note that there is no canonical BDD in $\mathrm{Restrict}(\mathcal{O}, f)$, since the BDDs in $\mathrm{Restrict}(\mathcal{O}, f)$ represent different functions.

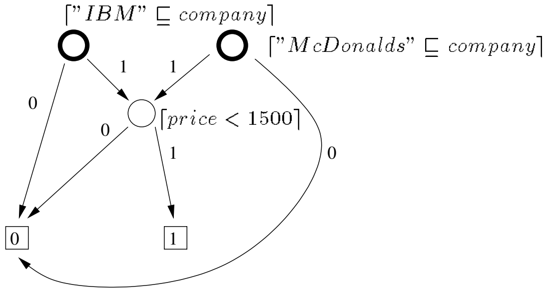

Figure 4: A query BDD.

Given two atoms $u, v \in A$, we write $u \models v$ to denote that every message which makes u true also makes v true. For example $v = 3 \models v > 1$.

Note that $v = 3$ and $v > 1$ are represented by two different BDD variables, although they are semantically related. In the following section, we deal with the question how to remove this redundancy from the BDD.

Eliminating Redundancies from Query BDDs

Note that different atoms in A may be semantically related because they contain information about the same attribute, but unrelated in the BDD. For example, $\lceil n = 10 \rceil$ and $\lceil n > 8 \rceil$ are different BDD variables. From the point of view of the BDD they are not related to each other. On the other hand, the definition of α_m guarantees that for all messages m, $\alpha_m(\lceil n = 10 \rceil) \rightarrow \alpha_m(\lceil n > 8 \rceil)$.

The semantic relation between BDD variables is formally expressed by the *dependency function* $d(A)$. The dependency function is given by

$$\bigwedge_{\{u,v\} \subseteq A, u \models v} \lceil u \rceil \rightarrow \lceil v \rceil \quad \text{AND} \quad \bigwedge_{\{u,v\} \subseteq A, u \models \neg v} \lceil u \rceil \rightarrow \neg \lceil v \rceil$$

$$\text{AND} \quad \bigwedge_{\{u,v\} \subseteq A, \neg u \models v} \neg \lceil u \rceil \rightarrow \lceil v \rceil.$$

We will use $d(A)$ later as a restriction function for BDDs.

BDD based query matching

In this section, we first treat the simpler case of total messages, for which all proposed semantics coincide. First, we show how the BDD evaluation algorithm can be applied to evaluate queries. Then, we describe how BDD restrictions can be used to improve the algorithm. Finally, we describe the evaluation strategies for other semantics.

SiSl. For this semantics one can use the procedure **EvalBDD** for query matching. Assume that $\mathcal{Q} = \langle \mathcal{Q}_1, \ldots, \mathcal{Q}_n \rangle$ is a sequence of queries or subscriptions. Given a total message m, let α_m be the assignment corresponding to the message. Now we execute the procedure **EvalBDD** on the parameters $\mathcal{Q}$ and α_m. A query Q_i is considered *matched* if the BDD node corresponding to Q_i evaluates to 1. Since the procedure **EvalBDD** works backward and because of the sharing between BDDs, the algorithm only evaluates a BDD

```
    Algorithm MVEvalBDD(O, α)
1   for v := n downto 1
2     if v is terminal
3       then value[v] := label[v]
4       else if α(label[v]) ≠ ⋆
5         then value[v] :=
              ITE⋆(α(label[v]), value[low[v]], value[high[v]])
6         else value[v] := ⋆
7   output value
```

Figure 5: Three-valued BDD Evaluation Algorithm

node once, or in other words the sub-query corresponding to a BDD node is evaluated only once. Moreover, the query BDD $\mathrm{BDD}(\mathcal{Q}, \prec)$ can be replaced by an arbitrary BDD from the set $\mathrm{Restrict}(\mathrm{BDD}_{\prec}(\mathcal{Q}), d(A))$. Since $d(A)$ is the dependency function, such a BDD is called a *dependency restriction* of $\mathrm{BDD}(\mathcal{Q}, \prec)$. BDD packages usually have good heuristics to compute small restrictions. Indeed, we shall see in Section 8 using restricted BDDs significantly improves the performance of the filtering engine. Correctness of this algorithm easily follows from the semantics.

DeSL. As we have shown in section 4, default semantics essentially amounts to handling total messages. When a message is received, it is extended into a total message. This operation can be implemented very effectively by using a default template whose attributes are changed by an incoming message. Therefore, default semantics can be handled by the algorithm **EvalBDD**.

StSL. For strict semantics, we present a BDD-based evaluation algorithm which performs significantly faster than the ordinary evaluation algorithm. Recall that in the strict semantics, a message m matches a subscription or query Q iff m is adequate for Q (all attributes occurring in Q are defined in m) and m satisfies Q.

The main idea is that an undefined atom renders all subformulas in which it occurs undefined. In the evaluation algorithm, we treat $\star$ as a third value denoting *undefined*. Whenever the label $\star$ is encountered, the algorithm **MVEvalBDD** of Figure 5 moves on to the next node. Otherwise, the truth value of the node is computed from its successors using the function $\mathrm{ITE}^{\star}$ which coincides with ITE for Boolean inputs, and evaluates to $\star$ otherwise.

Our experiments in Section 8 show that **MVEvalBDD** outperforms the algorithms for the other semantics significantly. This can be explained as follows:

(i) **EvalBDD** loops through all nodes. At each step, it accesses three different nodes. Thus, $3n$ node access operations are necessary.

(ii) **MVEvalBDD** on the other hand does not access the successor nodes of an undefined node. Thus, for messages which only defines i attributes, only i ($n \leq i \leq 3n$) nodes have to be accessed, and thus in the case of partial messages the constant factor of the algorithm is improved.

(iii) Even more importantly, not all node access operations are equally expensive. As long as the BDD is traversed from node n to node 1, consecutive memory parts are copied into cache memory, and thus, few cache failures occur. Therefore, consecutive node access operations are fast. In contrast, the successor nodes $low[v]$ and $high[v]$ of a node v are often not in the cache, and therefore, access operations to successor nodes are on average much more expensive. Therefore, the expected cost of the operation in line 5 of the program code in Figure 5 is much higher than the expected cost of the operation in line 6 of the program. Since the number of defined attributes in a message is correlated to the frequency of executing lines 5 and 6 respectively, it follows that the constant factor of **MVEvalBDD** is small for messages with few attributes.The experiments of Section 8 confirm these intuitions. It is easy to prove the correctness of the algorithm if $\star$ is interpreted as false.

7 OPTIMIZATION

In this section, we investigate natural optimization issues for three critical parts of the BDD filtering engine, in particular the BDD restriction, BDD variable ordering, and the BDD evaluation algorithm.

BDD Restriction. The Colorado BDD package [26] uses the restriction procedure described in [13] to compute restricted BDD.Note that this procedure is heuristic, and does not necessarily compute the smallest restriction of the BDD. Nevertheless, the performance of the restriction procedure depends strongly on the size of the restricting formula. Our experiments have shown that it is therefore not feasible to use the dependency function $d(A)$. Instead, we consider the dependency functions for each query *separately*, and perform iterative restrictions with these dependency functions. Our experiments clearly demonstrate the effectiveness of this procedure.

BDD Variable Ordering. It is well-known that for general BDDs, the variable order has a tremendous influence on BDD size. Since the problem of finding an optimal variable ordering is NP-hard [4], BDD packages either use complicated heuristics to determine a variable order, or let the user choose the variable ordering.

Our experiments have shown however that for the filtering engine application, the variable ordering does not have a strong influence on the BDD size, apparently because the boolean functions we consider are shallow, i.e., structurally simple, but numerous. In particular, the natural variable ordering where either BDD variables corresponding to the same attribute are kept close together, or variables are ordered according to frequency, both give good results; however they do not show significant improvement over random orders.

The stability of the algorithm with respect to variable ordering is an important feature of the filtering engine, since the shared BDD is extended *online* every time a new subscription arrives. It would be extremely expensive to adapt the variable order for each new subscription.

BDD Evaluation Algorithm. Recall that the major advantage of **MVEvalBDD** over **EvalBDD** was the avoidance of costly cache failures. In principle, it is possible to change the enumeration order of BDD nodes in such a way that the number of cache misses is minimized. Given the large number of BDD nodes, an optimal solution for this problem cannot be obtained in practice. Therefore, we have tried simple scheduling heuristics for changing the enumeration order of the BDD. So far, no significant performance improvements could be achieved in this way. The apparent reason is that the cache size of standard PC architectures is too small for this purpose.

8 EXPERIMENTAL RESULTS

All our experiments were carried out on a 200Mhz PentiumPro machine with 1GB RAM running RedHat linux (kernel 2.0.36). Our implementation of the filtering engine used the Colorado University BDD package (CUDD) version 2.3.0 [26].

Query Generation. Queries were produced by an automatic query generator internally developed at Microsoft. The query generator outputs queries on the basis of an input grammar file, where the grammar is specified in an extended BNF form and allows the user to associate probabilities with the production rules. The queries are generated by application of the rules, non-deterministic choices being selected in accordance with the associated probabilities.

In our experiments, we used ten attribute variables. There were three variables of type `integer`, three variables of type `double`, and four variables of type `string`. The integer (and, respectively double) variables could be combined with four integer (and, respectively double) constants 10, 30, 50 and 70 (and, respectively 10.0, 30.0, 50.0 and 70.0), using any of six relational operators $=, \neq, <, \leq, >$ and $\geq$. The string variables could be combined with four string constants "aa", "bb", "a" and "b", using four relational operators $=, \neq$, *substring* and *superstring*. Hence there were a total of 208 atomic formulas.

Note: We executed the entire experiment seven times. The results that follow report the averages over the seven runs.

Query Characteristics and Memory Requirements. We used a set of queries, with 7.6 atomic formulas, and 7.7 relational operators on an average. The following table shows the average number of BDD nodes obtained for the queries, both before and after applying the restriction optimization described in section 7. The significant reduction clearly indicates the effectiveness of the restriction optimization. Note that the number BDD nodes determines memory consumption; in highly optimized BDD packages such as the CUDD package which we use each node requires only a few bytes of memory. As can be clearly seen from the table the number of BDD nodes scales almost linearly with the number of queries. The state of the art BDD packages are capable of managing billions of nodes indicates that the number of queries our technique can handle is scalable, and will not become the bottleneck.

# of Queries	# of Nodes before Restriction	# of Nodes after Restriction	Factor of Improvement
25,000	180,226	108,889	1.66
50,000	351,616	206,848	1.70
75,000	543,677	299,942	1.81
100,000	708,969	376,992	1.88

Message Generation. Messages were generated by randomly assigning values to the messages variables. For numeric attribute variables, the values were chosen randomly from the range [0 . . . 80]. For the string variable the values were selected randomly from a set of twenty alternatives of varying length.

Evaluation. We carried out experiments using the **SiSL** and the **StSl** semantics. Recall that **SiSL** requires messages to be total. On the other hand **StSL** can handle partial messages (a typical case in realistic pub-sub systems). The following table shows the average time in seconds required to match 1000 messages for different numbers of queries and message densities. Message density is a measure of how total the message is, i.e., the number of attribute variables the message defines. The last column shows the time in seconds required to match 1000 total messages with using the **SiSL** semantics. As can be seen clearly from the table partial messages can be matched very quickly.

# of	Message Density					**SiSL**
Queries	7.7%	15.4%	23.1%	30.8%	100%	100%
25,000	1.56	3.81	5.86	8.23	27.64	20.12
50,000	3.56	7.63	10.92	14.74	55.66	41.58
75,000	6.65	12.49	18.11	22.56	89.93	61.75
100,000	9.01	15.57	21.4	30.83	121.32	81.64

As can be seen, the performance of the StSL algorithm scales almost linearly with the message density. As expected for total messages the performance for the **SiSL** semantics is better than that for the **StSL** semantics. The main reason for this is that the **MVEvalBDD** algorithm has to inspect both the children for every BDD node visited while the **EvalBDD** algorithm has to inspect only one child. However, we believe that partial messages is a common case for realistic pub-sub systems.

Our experiments clearly show that modern servers hosting our filtering engine will be *capable of matching 1000 messages against 500,000 queries in 15.57 seconds*. We are assuming that a modern server has a 1Ghz clock with 4GB memory and typical message densities are around 15%.

9 CONCLUSION

There are two major contributions of this paper. First, we provide various semantics for a message matching a query or subscription. These semantics differ in their expressiveness and the efficiency of the filtering algorithm. Our BDD filtering engine supports all these semantics. We also presented a filtering algorithm based on BDDs which is suitable for large scale content-based pub-sub systems. As was clearly demonstrated by the experimental results, our filtering algorithm can easily handle a half million subscriptions.

We plan to extend our work in several directions. The *Extensible Markup Language* or XML [11] is increasingly becoming the de-facto standard for exchanging data between internet applications. We want to extend our algorithm so that it can be used in pub-sub systems based on XML related technologies. Specifically, messages will be XML instances and subscriptions will be expressed in an *XML query language* such as *XQL*. Current requirements for XQL can be found in *http://www.w3.org/TR/xmlquery-req*. XQL is a more expressive query language than the ones presented in this paper. We plan to also implement a significant infrastructure that will enable us to perform experiments for realistic settings.

ACKNOWLEDGMENTS

The authors are thankful to Greg Zelesnik and Jianjun Chen for comments on a draft of this paper. Yuan Lu and Poul Williams have contributed to discussions initiating this work.

REFERENCES

[1] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, 1999.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. Information flow based event distribution middleware. In *Middleware Workshop at the International Conference on Distributed Computing*, 1999.

[3] G. Banavar, T.D. Chandra, B. Mukherjee, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, 1999.

[4] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[6] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transaction on Computers*, 40:205–213, 1991.

[7] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Interfaces and algorithms for a wide-area event notification service. *University of Colorado, CS Dept.*, Technical Report CU-CS-888-99, 1999.

[8] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000)*, 2000.

[9] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Content-based addressing and routing: A general model and its application. *University of Colorado, CS Dept.*, Technical Report CU-CS-902-00, 2000.

[10] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[11] W3C consortium. Extensible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml, February 1998.

[12] The Common Object Request Broker: Architecture and specification. OMG Document Number 91.12.1, December 1991. Revision 1.1 (Draft 10).

[13] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *IEEE International Conference on Computer-Aided Design (ICCAD'90)*, pages 126–129, 1990.

[14] G. Fitzpatrick, T. Mansfield, S. Kaplan, D. Arnold, T. Phelps, and B. Segall. Augmenting the workaday world with elvin. In *Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)*, 1999.

[15] D. Garlan, G.E. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE Computer*, 25(6), June 1992.

[16] C. Gerety. HP Softbench: A new generation of software development tools. Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, November 1989.

[17] J. Gough and G. Smith. Efficient recognition of events in a distributed system. In *18th Australasian Computer Science Conference (ACSC18)*, 1995.

[18] Gryphon homepage. http://www.research.ibm.com/gryphon.

[19] H. Jubin. *Javabeans by example*. Upper Saddle River: Prentice Hall, 1998.

[20] Keryx homepage. http://keryxsoft.hpl.hp.com.

[21] C. Low. Integrating communication services. *IEEE Communications Magazine*, 35(6), 1997.

[22] Sun Microsystems. Java message service, version 1.0.2. http://www.javasoft.com, November 1999.

[23] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Int. Conf. on Cooperative Information Systems (COOPIS)*, 2000.

[24] S.P. Reiss. Connecting tools using message passing in the FIELD program development environment. *IEEE Software*, July 1990.

[25] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *AUUG 97*, 1997.

[26] Fabio Somenzi. CUDD: CU Decision Diagram Package. Available from http://vlsi.colorado.edu/ fabio/.

[27] K. Sullivan and D. Notkin. Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology*, 1(3), July 1992.

[28] SunSoft. *Tooltalk 1.1.1 Users's Guide*, November 1993.