Diagnosis of Asynchronous Discrete Event Systems: Datalog to the Rescue!*

Serge Abiteboul
INRIA-Futurs& U. Paris Sud
<fname>.<lname>@inria.fr

Zoë Abrams
Stanford U.
zoea@stanford.edu

Stefan Haar
INRIA Rennes
stefan.haar@irisa.fr

Tova Milo
Tel Aviv U.
milo@cs.tau.ac.il

ABSTRACT

We consider query optimization techniques for data intensive P2P applications. We show how to adapt an old technique from deductive databases, namely Query-Sub-Query (QSQ), to a setting where autonomous and distributed peers share large volumes of interelated data.

We illustrate the technique with an important telecommunication problem, the diagnosis of distributed telecom systems. We show that (i) the problem can be modeled using Datalog programs, and (ii) it can benefit from the large battery of optimization techniques developed for Datalog. In particular, we show that a simple generic use of the extension of QSQ achieves an optimization as good as that previously provided by dedicated diagnosis algorithms. Furthermore, we show that it allows solving efficiently a much larger class of system analysis problems.

1. INTRODUCTION

Research on deductive databases, a hot topic in the late 80s, led to beautiful results, with little industrial impact. Years later, with networks everywhere, recursive data management is becoming more essential. For instance, telecommunication systems interact with each other to gather routing data, possibly recursively. Also, a Web portal may want to retrieve information from some Web servers, referring to other servers recursively. In both cases, recursive data management is an essential aspect of the problem. Things are of course more complex than in the good old Datalog days: the architecture is often based on distributed autonomous peers and the interaction is often asynchronous. Nevertheless, one encounters again the management of a mix of intensional and extensional information in a recursive setting. As we will see here, some of the solid technology developed for Datalog comes in handy.

*This research has been conducted while the second and fourth authors were visiting INRIA-Futurs. The research has been partially supported by the European Project EDOS, the RNRT-SWAN project, the ACI project MDP2P, the Arc ASAX, the France-Stanford Center for Interdisciplinary Studies and the Israel Science Foundation.

PODS 2005 June 13-15, 2005, Baltimore, Maryland.

Telecom systems. To illustrate issues often encountered in data intensive P2P applications, we consider in this paper a particular application, the diagnosis of asynchronous discrete event systems. We will show that although, on the face of it, this does not appear to be a pure database application, (i) the problem can be suitably modeled using Datalog programs, and (ii) it can benefit from the large battery of optimization techniques developed for Datalog.

A *telecommunication network* consists of a large number of peers (pieces of hardware and software) that are distributed. Each peer runs some application that may fail in various occasions and that issues, depending on its state, *alarm signals*. The operational logic of each peer, including the possible states and their corresponding alarm signal, is described by a Petri net. Each local peer has only a partial view of the system, and its local time is not synchronized with that of the other peers. Alarms are reported to supervisors that perform *diagnosis*; we will consider the case with a single supervisor only. The communication of alarms over the network causes a loss of synchronization that results in the need to consider their nondeterministic interleaving at the supervisor to analyze the origin of the fault. Suppose that the system fails (i.e. some kind of severe fault is reported); the supervisor needs to determine what actually happened in the global system. Observe that the information available when performing this analysis may be seen as partially *extensional* data (e.g., a sequence of alarms received by the supervisor), and partially *intentional* (e.g., the possible execution flow of some peer as described by its Petri net). Also observe the presence of recursion since the execution in one peer may depend on the execution at some other peers, and vice versa.

We will show that diagnosis problems can be stated in terms of query evaluation in deductive databases.

Naturally a main concern is the efficiency of the diagnosis process. Typically, one examines the possible executions and isolates those that correspond to what was observed by the supervisor, e.g., perform some analysis of the "unfoldings", see [13], of Petri nets. A technique is proposed in [8] to reduce the portions of the unfoldings that are constructed during this analysis. Similarly, for Datalog, two main, closely related, optimization techniques for query evaluation in deductive databases have been studied, namely Query-Sub-Query (QSQ) [34] and Magic Set [7] (among others), that both aim at minimizing the quantity of data that is materialized.

We will show how an extension of QSQ may be used to perform efficient diagnosis of asynchronous systems.

We enrich Datalog and QSQ to handle distribution, not a totally new concern; see, e.g., [19, 33]. The core of the QSQ technique consists in the rewriting of a Datalog program given a query. In the diagnosis context, the Datalog program (i.e. the alarms and the Petri nets) is distributed over several peers. We show that each peer can perform its own rewriting with only local information available.

We call our distributed extension of QSQ, dQSQ.

Interestingly, we will show that a simple "generic" use of dQSQ achieves as good an optimization as that previously provided by the dedicated diagnosis algorithm of [8]. Moreover, we will see that it allows optimizing a much larger class of system analysis problems, including situations where only part of the alarms are reported, or when alarm patterns need to be detected. As soon as the problem can be stated in Datalog terms (and we will see that this is possible for many important diagnosis problems), dQSQ can be applied to optimize the evaluation.

P2P information and ActiveXML. We believe that beyond this particular application, deductive database techniques are well suited for a large range of applications that involve autonomous peers managing large volumes of data. For instance, the use of declarative queries for routing information in a network naturally leads to recursive query processing [16]. Indeed, the authors of the present paper have adapted QSQ to a P2P setting while working on query optimization for ActiveXML [5], i.e., XML documents where some of the data is given intentionally by means of calls to Web services. The analogy between deductive databases and active documents is quite strong. Think of an element node in a document as a collection. This collection may be given extensionally or specified intensionally with a call to some Web service. So such nodes play the role of a Datalog predicate and calls to Web services, the role of Datalog rules. The recursion comes in naturally: think of a Web service calling a second Web service that calls the first.

The *dQSQ* optimization technique presented here is in fact a subset of a technique developped for ActiveXML, namely *axmlQSQ*. The goal was to dramatically reduce data materialization and communication via the management of selective service invocation, service refinement and sideway information passing. axmlQSQ is substantially more complex than dQSQ because it also has to address issues related to tree manipulation and node creation (when trees are copied). A difficulty in the tree setting is query *pushing* into services, that entails some form of XML query composition and tree unification. An implementation is on-going that is based on asynchronous exchange of information flows between the peers involved in a computation.

In the late 80's, deductive databases had many fans who (we think correctly) believed that recursive query evaluation techniques were capturing fundamental aspects of information management. It turned out that the addition of transitive closure to relational systems was sufficient to handle most applications that were considered at that time. So, deductive databases did not convince industry. We believe that the management of large amounts of data in distributed peers mutually depending of each other (as in the diagnosis problem and more generally in ActiveXML settings), naturally motivates the use of the deductive database paradigm.

The paper is organized as follows. Section 2 defines the diagnosis problem. The distributed dDatalog and dQSQ are presented in Section 3, and applied to the diagnosis problem in Section 4. Related works are discussed in Section 5.

2. PRELIMINARIES

We start by formally defining the diagnosis problem studied. The model that we use is typical for modeling telecommunication systems, and is used in particular in the framework of the Swan project [31]. A *Petri Net* describes the behavior of each peer in a P2P system and the alarms each of them emits depending on its state. The *unfolding* of the Petri net is a representation of the (possibly infinite) set of possible runs for such a system. An *alarm sequence* describes a sequence of alarms gathered by the system supervisor.

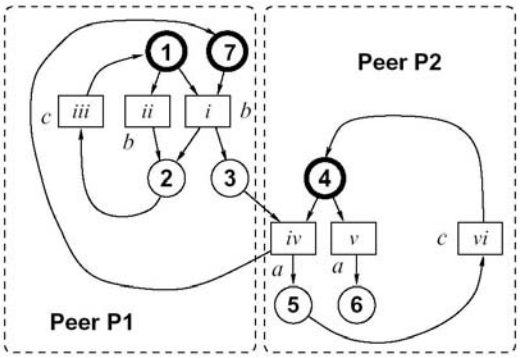

Figure 1: A Petri net.

To formalize this, let $\mathcal{V}$ and $\mathcal{A}$ be infinite domains of *nodes* and *alarm symbols* respectively. Let $\mathcal{P}$ be an infinite domain of *peer names*. The two notions of *net* and *Petri net* given next are central for describing the problem.

DEFINITION 1. *A* net *is a directed labeled graph* $N = (S, T, E, \alpha, \phi)$. *$S, T \subseteq \mathcal{V}$ are two disjoint (possibly infinite) sets of nodes, called* places *and* transitions, *resp. $E \subseteq (S \times T) \cup (T \times S)$ is a set of edges, connecting places and transition nodes. $\alpha : T \to \mathcal{A}$ and $\phi : S \cup T \to \mathcal{P}$ are two node labeling functions. α associates an alarm symbol to each transition in the net, and ϕ associates a peer name to each place and transition in the net. For $v \in S \cup T$, we denote by ${}^\bullet v = \{u \mid (u, v) \in E\}$ the* parents *of node v, and by $v^\bullet = \{u \mid (v, u) \in E\}$ its* children.

DEFINITION 2. *A* Petri net (N, M) *consists of a net $N = (S, T, E, \alpha, \phi)$ where S and T are finite, and a distinguished subset $M \subseteq S$ of the places of N, called the* marked places *of the net. A transition node t of the Petri net is* enabled *iff all its parent nodes are marked. Such a transition can* fire *and yield a new Petri net (N, M') where $M' = M - {}^\bullet t + t^\bullet$; we assume Petri nets are* safe, *i.e. if t is enabled in some reachable marking M, then $M \cap t^\bullet = \emptyset$.*

We will use as a running example the safe Petri net given in Figure 1. Transitions are denoted by squares and places by circles. The marked places are in bold. The numbers inside the nodes are the node identifiers. The labels next to transition nodes are their associated alarm symbols. We have two peers $P1$ and $P2$. For instance, $\alpha(i) = b$, $\phi(i) = P1$, ${}^\bullet i = \{1, 7\}$, $i^\bullet = \{2, 3\}$. Transition i, ii and v are enabled. If transition i fires, the marking from places $1, 7$ is removed and places $2, 3$ become marked. The markings of the places in a particular peer model the current state of this peer, and the transition nodes capture possible state transitions. Petri nets generalize automata, in the sense that global states decompose into several local states represented by marked places. Several transitions may occur in parallel or in any order: this will be referred to as *concurrency*.

An *execution* of a Petri net is a sequence of firings of transitions, up to interleaving (i.e. permutation of pairwise independent events). *Unfoldings* of Petri nets (see Figure 2) are representations of all possible executions, in the form of a particular (acyclic) net together with a *net homomorphism*, defined next, to the original Petri net.

DEFINITION 3. *A* homomorphism *from a net N to a net N' is a mapping $\rho : S \cup T \to S' \cup T'$ preserving the peer, the alarm*

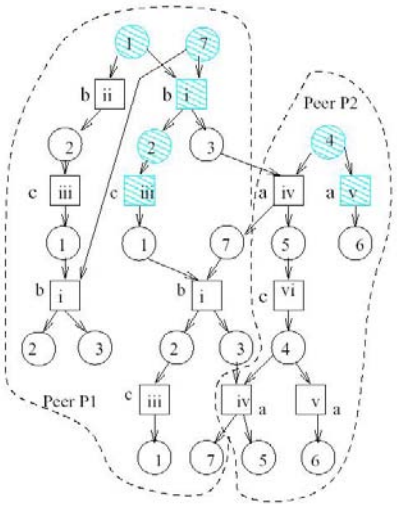

Figure 2: A branching process of a Petri net.

symbol, and the type (i.e. place *or* transition*) of each node. Furthermore, for every place $v \in S$, the restriction of ρ to ${}^\bullet v$ (and resp.$v^\bullet$) is a bijection from ${}^\bullet v$ onto ${}^\bullet\rho(v)$ (resp. $v^\bullet$ onto $\rho(v)^\bullet$).*

The fact that one transition may disable or enable another is captured in the unfolding by the *conflict* and *causal* relationships among the nodes, respectively.

DEFINITION 4. *Two nodes v and u of a net N are in* causal relation*, denoted by $v \preceq u$, iff either $v = u$ or the net contains a path from v to u. Two nodes v and u of a net N are in* conflict relation*, denoted $v\#u$, iff N contains two distinct transition nodes t, t' with a common parent and where $t \preceq v$ and $t' \preceq u$. If neither $v \preceq u$ nor $u \preceq v$ nor $v\#u$, u and v are* concurrent*, written $u \parallel v$. An* unfolding *or* branching process *of a Petri net (N, M) is a pair $U = (\widehat{N}, \rho)$ where $\widehat{N}$ is a (possibly infinite) net and ρ a homomorphism $\widehat{N} \to N$ such that:*

- *$\widehat{N}$ is acyclic (i.e. $\preceq$ is a partial order), and for each node v in it, the set $\{u \mid u \preceq v\}$ is finite.*
- *the image under ρ of the set c_0 of roots of $\widehat{N}$ is the set M of marked places in (N, M).*
- *no node has two conflicting parents.*
- *each place node in $\widehat{N}$ has at most one incoming edge,*
- *for every two distinct transition nodes $t, t' \in \widehat{N}$, either ${}^\bullet t \neq {}^\bullet t'$ or $\rho(t) \neq \rho(t')$.*

A configuration *C of $\widehat{N}$ is a set of nodes containing c_0,* downward closed *($v \in \kappa$ and $u \preceq v$ imply $u \in C$) and* conflict-free *($v \in C$ and $u\#v$ imply $u \notin C$).*

A branching process of the Petri Net of Figure 1 is represented in Figure 2 (ignore for now the shading of some nodes). The places / transitions there represent instances of the corresponding Petri net places / transitions visited during a run of the system[1]; the ids in the nodes refer to the Petri net nodes they map to. The alarm next to a transition is the alarm of the corresponding transition in the Petri net.

The set of all branching processes of Petri net (N, M) is uniquely defined up to isomorphism. For two branching processes U, U' of a Petri net (N, M), U' is a *prefix* of U written $U' \sqsubseteq U$, if U's node set is contained in that of U (note that unfoldings generate downward closed sets, thus the name "prefix" is justified). By [13], there is a unique (up to isomorphism) $\sqsubseteq$-maximal branching process. $Unfold(N, M)$, called *the unfolding of* (N, M).

The problem. When a transition fires, an alarm corresponding to the associated alarm symbol is sent to the supervisor. We model such an *alarm* as a pair (a, p) where a is some alarm symbol and p is the peer that emitted this alarm. An *alarm sequence* received by the supervisor is thus a sequence $(a_1, p_1), (a_2, p_2) \ldots (a_n, p_n)$. Recall that we assume asynchronous communications, so we do not guarantee that the alarms sent by different peers appear in the order they were emitted; we can only assume that for each individual peer the relative order of its alarms in the sequence respects the order in which they were sent by the peer. The goal is to find an explanation for a given alarm sequence.

Input:. A Petri net (N, M) and an alarm sequence $A = (a_1, p_1), (a_2, p_2) \ldots (a_n, p_n)$.

Output:. All configurations C of $Unfold(N, M)$ s.t. there is a bijection τ from the alarms in A to the alarms in C that (i) preserves the alarms symbol (i.e. $\alpha(\tau(a_i, p_i)) = a_i$), (ii) preserves the peer names (i.e., $\phi(\tau(a_i, p_i) = p_i)$, and (iii) does not contradict the partial order of alarms for this particular peer (i.e., for each $(a_i, p_i), (a_j, p_j), i < j$, if $p_i = p_j$ then it cannot be the case that $\tau(a_j, p_j) \preceq \tau(a_i, p_i)$. We call this set the *diagnosis set* (of A in (N, M)). In practice, this set will have to be "explained" to a human supervisor and represented (preferably graphically) in a compact form.

To continue with our example, the set of shaded nodes in Figure 2 is a diagnosis (i. e. configuration giving a possible explanation) for the alarm sequence (b, p_1), (a, p_2), (c, p_1). The same set of nodes is also a diagnosis for the alarm sequence (b, p_1), (c, p_1), (a, p_2), but not, for instance, for (c, p_1), (b, p_1), (a, p_2).

3. DISTRIBUTED DATALOG AND QSQ

We will model Petri net unfoldings and alarm sequences using a distributed version of Datalog, that we call $dDatalog$, and optimize it with a distributed version of QSQ, that we call dQSQ. We define these next, borrowing notation from [32]. A main difference from [32] is that peer names here are constants, while they are allowed to be variables in [32]. The departure from classical Datalog is that we allow the presence of function symbols. This is needed to capture the creation of the nodes when constructing the unfolding. Note that, as a consequence, the semantics of a Datalog program may be infinite and its naive evaluation may not terminate.

Syntax. We assume infinite domains $\mathcal{D}$ of constants, Var of variable names, $\mathcal{F}$ of function names, and a fixed collection of relation symbols $R_1, \ldots, R_n$. We use x, y, z for variables, b, c, p for constants, f, g for functions, and e for terms constructed by applying functions on constants, variables, and other terms. Terms that contain no variables are called *ground*. An *atom* a has the form $R@p(e_1, \ldots, e_n)$ where p is a constant (representing a peer name). The intuition is that $R(e_1, \ldots, e_n)$ "holds" at peer p.

A *rule* has the form

[1]In the Petri net literature, the unfolding instances of the Petri net *places* (resp. *transitions*) are often called conditions (resp. *events*). To minimize the terminology introduced in the paper, we refer to both as places (resp. transitions), keeping in mind that one place in the unfolding is an instance of one in the Petri net, and similarly for a transition.

$\underline{r}$	*relations*	R, A		
	$\underline{rule\ 1}$	$R@r(x,y)$	:-	$A@r(x,y)$
	$\underline{rule\ 2}$	$R@r(x,y)$	:-	$S@s(x,z), T@t(z,y)$
$\underline{s}$	*relations*	S, B		
	$\underline{rule\ 3}$	$S@s(x,y)$	:-	$R@r(x,y), B@s(y,z)$
$\underline{t}$	*relations*	T, C		
	$\underline{rule\ 4}$	$T@t(x,y)$	:-	$C@t(x,y)$

Figure 3: A dDatalog program

$$a_0 \text{ :- } a_1, \ldots, a_n, x_1 \neq y_1, \ldots, x_m \neq y_m$$

where the a_i's, $i = 0 \ldots n$, are atoms and the x_j, y_j, $j = 1 \ldots m$ are constants or variables names that appear in $a_1 \ldots, a_n$. Following standard terminology, a_0 is the head, $a_1, \ldots, a_n, x_1 \neq y_1, \ldots, x_m \neq y_m$ is the body, and when $n = 0$, a_0 is called a *fact*. We require that all the variables in the head appear also in the body of the rule.

A program P is a finite set of rules. The rules at site p are the rules where p is the site of the head. The intuition is that peer p holds the rules defining relation $R@p$. When all the atoms appearing in P belong to one site, say p, we say that P is a *local* program. For such local programs, when the peer p is clear from the context, we will omit it and use a shorthand notation $R(e_1, \ldots, e_n)$ rather than $R@p(e_1, \ldots, e_n)$.

As an example, consider the dDatalog program in Figure 3. The rules are distributed between three peers, r (hosting relations R, A), s (hosting S, B) and t (hosting T, C). Relations R, S, T are intensional, i.e., defined by the program; and A, B, C are base relations, i.e., given extensionally as facts. For defining queries, we also use rules, e.g., $Q@r(y)$:- $R@r(\text{“1”}, y)$. This query, posed at peer r, computes the R tuples having the value “1” in their first column and projects out the first column.

Models and Semantics. There is a canonical translation of a dDatalog program P into a Datalog program P^g called the *global Datalog program*: each n-ary relation name R is translated into an $(n+1)$-ary relation name R^g, and each atom $R@p(t_1, \ldots, t_n)$ into $R^g(t_1, \ldots, t_n, p)$. We define a *model* of P to be a model of P^g. As usual the semantics of P is its minimal model.

3.1 Query evaluation

We start by considering the evaluation of local programs, and then move to distributed ones. We first consider naive query evaluation and then its optimization with QSQ.

Naive evaluation revisited. We differ from standard naive evaluation of Datalog in that we want to think of a naive evaluation as a continuous flow of tuples. The evaluation works as follows. It starts with the query relation and *activates* it. When a relation is activated, it activates all rules defining it. When a rule is activated, it activates all relations occurring in its body. A rule that is activated continuously receives tuples from the relations in its body and produces tuples (by evaluating the corresponding query). The computation terminates when (1) no more new rule or relation may be activated and (2) no new fact may be derived by any of the activated rules. This evaluation is slightly different but equivalent to classic naive evaluations.

Query-sub-Query. QSQ is a beautiful but complex technique and for space limitation, we do not intend to present it here in detail, see [34, 3]. However, our work extends QSQ, so we give some intuition on how it works.

$\underline{Query}$	$Q(x)$	:-	$R^{bf}(\text{“1”}, x)$
	$in\text{-}R^{bf}(\text{“1”})$	:-	
$\underline{Rule\ 1}$	$sup_{10}(x)$	:-	$in\text{-}R^{bf}(x)$
	$sup_{11}(x,y)$	:-	$sup_{10}(x), A(x,y)$
	$R^{bf}(x,y)$	:-	$sup_{11}(x,y)$
$\underline{Rule2}$	$sup_{20}(x)$	:-	$in\text{-}R^{bf}(x)$
	$sup_{21}(x,y)$	:-	$sup_{20}(x), S^{bf}(x,y)$
	$sup_{22}(x,y)$	:-	$sup_{21}(x,z), T^{bf}(z,y)$
	$in\text{-}S^{bf}(x)$	:-	$sup_{20}(x)$
	$in\text{-}T^{bf}(y)$	:-	$sup_{21}(x,y)$
	$R^{bf}(x,y)$	:-	$sup_{22}(x,y)$
$\underline{Rule\ 3}$	$sup_{30}(x)$	:-	$in\text{-}S^{bf}(x)$
	$sup_{31}(x,y)$	:-	$sup_{30}(x), R^{bf}(x,y)$
	$sup_{32}(x,y)$	:-	$sup_{31}(x,y), B(y,z)$
	$in\text{-}R^{bf}(x)$	:-	$sup_{30}(x)$
	$S^{bf}(x,y)$	:-	$sup_{32}(x,y)$
$\underline{Rule\ 4}$	$sup_{40}(x)$	:-	$in\text{-}T^{bf}(x)$
	$sup_{41}(x,y)$	:-	$sup_{40}(x), C(x,z)$
	$T^{bf}(x,y)$	:-	$sup_{41}(x,y)$

Figure 4: The QSQ rewriting of the Datalog program

The crux of the QSQ optimization technique is to minimize the number of tuples derived. This is achieved by a rewriting of the program based on the propagation of bindings. QSQ starts from the rule defining the query. It processes the body of the rule from left to right. For each atom in the rule, it creates a ”supplementary relation” to keep the bindings of variables that can be used for this atom. It obtains a new subquery, namely a call to the relation of this atom with the binding provided by the supplementary relation. More precisely, the QSQ rewriting is based on *binding patterns* and *supplementary relations*.

Binding Patterns. For each relation name, consider *adorned versions* of the relation based on the bindings of the variables: e.g., for the 2-ary relation R above, R^{bb}, R^{bf}, R^{fb} and R^{ff} represent, respectively, the case with both variables bound, only the first one bound, only the second, and none. The top down, left-to-right evaluation of the rules determines the propagation of bindings.

Supplementary relations. For each adorned relation and each position in the body of an adorned rule, a *supplementary relation* is introduced to accumulate the bindings relevant to that position. The notation $sup_{i,j}$ is used to denote a supplementary relation for position j in rule i.

For instance, consider a local version P_{local} of the program in Figure 3, ignoring the locations of rules and relations and assuming they all reside on one peer. The QSQ rewriting of P_{local} is given in Figure 4.

The QSQ evaluation has nice properties. It computes the correct answer to the query. It materializes only a *minimal* set of tuples. It is guaranteed to terminate when the program contains no function symbol. Clearly, QSQ deserves more elaboration and we refer readers not familiar with this technique to [34, 3].

3.2 From QSQ to distributed QSQ

We next adapt QSQ to a distributed setting.

$\underline{r}$ $Q@r(x)$:- $R^{bf}@r(\text{“1”}, x)$
$in\text{-}R^{bf}@r(\text{“1”})$:-
$sup_{10}@r(x)$:- $in\text{-}R^{bf}@r(x)$
$sup_{11}@r(x,y)$:- $sup_{10}@r(x), A(x,y)$
$R^{bf}@r(x,y)$:- $sup_{11}@r(x,y)$
$sup_{20}@r(x)$:- $in\text{-}R^{bf}@r(x)$
$R^{bf}@r(x,y)$:- $sup_{22}@s(x,y)$
$sup_{31}@r(x,y)$:- $sup_{30}@s(x), R^{bf}@r(x,y)$
$in\text{-}R^{bf}@r(x)$:- $sup_{30}@s(x)$
$\mathbf{sup_{32}@r(x,y)}$:- $\mathbf{sup_{32}@s(x,y)}$

$\underline{s}$ $sup_{21}@s(x,y)$:- $sup_{20}@r(x), S^{bf}@r(x,y)$
$in\text{-}S^{bf}@s(x)$:- $sup_{20}@r(x)$
$\mathbf{sup_{22}@s(x,y)}$:- $\mathbf{sup_{22}@t(x,z)}$
$sup_{30}@s(x)$:- $in\text{-}S^{bf}@s(x)$
$S^{bf}@s(x,y)$:- $sup_{32}@r(x,y)$
$sup_{32}@s(x,y)$:- $sup_{31}@r(x,y), B@s(y,z)$

$\underline{t}$ $sup_{22}@t(x,y)$:- $sup_{21}@s(x,z), T^{bf}@t(z,(y))$
$in\text{-}T^{bf}@t(y)$:- $sup_{21}@s(x,y)$
$sup_{40}@t(x)$:- $in\text{-}T^{bf}@t(x)$
$sup_{41}@t(x,y)$:- $sup_{40}@t(x), C@t(x,z)$
$T^{bf}@t(x,y)$:- $sup_{41}@t(x,y)$

Figure 5: The full distributed QSQ rewriting

Naive distributed evaluation. Let us first reconsider newly activated relations in our naive query evaluation. For local relations, the treatment is the same as before. For external relations, a request has to be sent to the external site. Then tuples start being produced in various sites and exchanged. The system reaches a fixpoint when no new relation may be activated and no new fact derived *at any peer*. It is easy to see that the result is exactly as in the centralized case. One problem here is the detection of *termination.* Since the state of the Datalog program is distributed, it is more complex to detect termination than in classical Datalog. Nevertheless one can use standard termination detection algorithms for distributed computing, in the style of [19, 33], for detecting that all peers are in idle mode; details omitted.

Distributed QSQ (dQSQ). The dQSQ processing starts at the peer where the query is posed. As in centralized QSQ, we start with the rule defining the query, and then in a top-down fashion, process the body of rules defining each encountered relation, from left to right. The only difference is that now, when a remote relation is encountered, the peer delegates the processing of the remainder of the rule (from the remote relation name to the right end of the rule) to the remote peer in charge of that relation.

To illustrate the process, assume that the query $Q@r(y)$:- $R@r(\text{“1”}, y)$ is posed to peer r. Peer r rewrites its own rules (*Rule 1* and *Rule 2*). The rewriting of the first is like in the local case since it contains no remote relations. For the second rule, when the rewriting encounters the remote relation S, it sends to peer s the remainder of the rule, namely a query that defines the bindings that need to be computed by the remainder of the rule:
(†) $sup_{22}@s(x,y)$:- $sup_{20}@r(x), S^{bf}@s(x,z), T^{bf}@t(z,y)$

Peer s processes (†), again using a QSQ rewriting. Observe that, if a peer receives the same request from different peers , it reuses the same *machinery* to answer.

To conclude, Figure 5 gives the resulting dQSQ program. Observe that it is almost identical to that of Figure 4, obtained in the local case. The only difference is that some supplementary relations are defined in one peer and then sent to another peer where their bindings are used. (See rules in bold for sup_{22} and sup_{32}.) In general, one can verify that $dQSQ$ is as optimal as QSQ, namely materializes the same minimal information.

THEOREM 1. *Let P be a dDatalog program, and assume w.l.o.g. that the relation names of distinct peers are different*[2]*. Let P_{local} be a "local" version of P ignoring peer names, $\widehat{P}$, $\widehat{P}_{local}$ be the rewritten programs generated from P and P_{local} by the dQSQ and QSQ algorithms, resp.*

1. *There is a surjection ζ from the relation names in $\widehat{P}$ to those of $\widehat{P}_{local}$, that is a bijection for all the adorned relations, and where every relation $R@p \in \widehat{P}$, $\widehat{P} \models R@p(c_1, \ldots, c_n)$ iff $\widehat{P}_{local} \models \zeta(R)(c_1, \ldots, c_n)$.*
2. *dQSQ computes the same facts (up to the mapping ζ) as QSQ and terminates on P iff QSQ does on P_{local}.*

An important point is that in dQSQ *the rewriting is performed locally at each peer without any global knowledge.*

We conclude with two general observations on dQSQ.

Remark 1. One could use a different distribution for the supplementary relations, based on some cost model.

Remark 2. The dQSQ computation, and the generation of results, may start even before the rewriting is complete. This property is especially important in the context of the Web where the number of sites transitively involved in a computation may be too large to explore exhaustively.

4. FROM PETRI NETS TO DQSQ

In this section, we show how to use dDatalog and dQSQ to solve the diagnosis problem. We first explain how unfoldings and alarm sequences are modeled in dDatalog. Then we show the powerful effect of dQSQ in this context. Naturally there are different ways to express the problem in dDatalog. We first give a simplified version of the program, then explain how it can be improved.

4.1 Unfolding as dDatalog

We now explain how the Petri net unfolding can be expressed using dDatalog. Observe that the rules at each peer are defined *locally* at the peer: They are based solely on the *peer's view of the Petri net*, namely the places and transitions of the peer and their nearby neighborhood *without any global knowledge of the overall net structure.*

To simplify, we assume below that every transition node has exactly two parents. This can be generalized to the arbitrary case in a straightforward manner. We use further the constant symbols $c_1, \ldots, c_n$ to identify the nodes of the petri net. These correspond, for instance, in our running example to the node identifiers $1-7, i-iv$. W.l.o.g we assume that the node identifiers are all distinct [3].

We say that a peer p' is a *neighbor* of peer p, if it holds a transition t' that controls a place node that is used to fire some transition t in p, i.e., p' holds a transition node that is a grandparent of some transition in p. We denote by $Neighb(p)$ the set of

[2]Otherwise rename the relations, e.g. by concatenating the peer name.

[3]To guarantee that distinct peers use different identifiers the peer id can be concatenated.

neighbor peers of peer p. For instance, in our running example, $Neighb(p_1) = \{p_1, p_2\}$.

We define eight intensional relations at each peer p: $places$, $trans$, map, $notCausal$, *notConf*, $causal$, $placesTree$ and $transTree$. The first two relations $places$ and $trans$ describe respectively the place and transition nodes of the unfolding. Intuitively, the atom $trans(t, s, s')$ states that the transition node t is a child place of nodes s and s'. Similarly, $places(s, t)$ indicates that place node s is a child of transition node t. (Recall from the definition of the unfolding that each place node has a unique parent). The relation map represents the map ρ between the nodes of the unfolding and those of the Petri net. Namely, $map(x, c)$ holds for a node x of the unfolding and a node c of the Petri net if $\rho(x) = c$.

The $notCausal(x, y)$ and *notConf*(x, x, y) atoms represent the non causality and non conflict relation between the unfolding nodes x, y and are used for the construction of the $places$ and $trans$ of the unfolding, following Definition 4. Finally, $causal$, $placesTree$, and $transTree$ are auxiliary relations used for the definitions of the previous two. We next explain how each of these relations is defined.

trans, places, map:. To define these relations, we use two function symbols f and g to generate fresh identifiers for the unfolding nodes.

The first rule creates the unfolding roots. For each marked place node c_r of the Petri net in peer p, we have a rule creating a corresponding place node of the unfolding, and mapping it to c_r. We use some arbitrary virtual transition node id r as the "parent" of c_r.

$$(\dagger\dagger) \quad \begin{array}{ll} places@p(g(r, c_r), r) & :- \\ map@p(g(r, c_r), c_r) & :- \end{array}$$

We next build inductively the unfolding. For brevity, we use below the notation a, a':-*body* as a shorthand for the two rules a:-*body* and a':-*body* having the same body. For each transition node c in p with grandparent nodes at peers p', p'', peer p has the rule:

$$\begin{array}{ll} trans@p(f(c,u,v),u,v), & \\ map@p(f(c,u,v),c) \;\; :- & map@p'(u,c'), map@p''(v,c''), \\ & places@p'(u,u'), places@p''(v,v'), \\ & notCausal@p'(u',v), \\ & notCausal@p''(v',u), \\ & \textit{notConf}@p'(u',u',v') \end{array}$$

Note that the petri node ids c, c', c'' and the peer ids p, p', p'' in the above rules are all constants, which is in accordance with the dDatalog syntax that requires constant peer ids.

Similarly, for each place node c' in the Petri net that is a child of node c at peer p, we have the rule:

$$\begin{array}{ll} places@p(g(x,c'),x), & \\ map@p(g(x,c'),c') \quad :- & map@p(x,c), \\ & trans@p(x,y,z) \end{array}$$

We next explain how the $notCausal$ and *notConf* used in the above rules are defined.

notCausal:. This is the complement of the $causal$ relation. Interestingly, it can be defined positively. Namely, for two transition nodes, x, y in the unfolding, $notCausal(x, y)$ indicates that $\neg[y \preceq x]$. For all peer names $p', p'' \in Neighb(p)$, peer p holds the following rules:

$$\begin{array}{l} notCausal@p(x,y) \text{ :-} \\ \quad trans@p(x,u,v), places@p'(u,u'), \\ \quad notCausal@p'(u',y), places@p''(v,v'), \\ \quad notCausal@p''(v',y), u \neq y, v \neq y, x \neq y \end{array}$$

Additionally, we have to use one rule to state that the virtual transition node r (used in rule (††) above) is not causal to any transition node:

$$notCausal@p(r,x) \;\; :- \;\; trans@p(x,y,z)$$

To conclude, we define *notConf*. The definition uses three auxiliary relations $causal$, $placesTree$, and $transTree$.

causal:. For two transition nodes x, y of the unfolding, the atom $causal(x, y)$ indicates that node y is an ancestor of node x (i.e. $y \preceq x$). *causal* and *notCausal* are complements. However, since the emphasis is on negation free dDatalog, we define both directly in a positive manner. We will return to this issue later. For peers $p', p'' \in Neighb(p)$, peer p holds the following rules:

$$\begin{array}{lll} causal@p(x,y) & :- & trans@p(x,u,v), places@p'(u,y) \\ causal@p(x,y) & :- & trans@p(x,u,v), places@p''(v,y) \\ causal@p(x,y) & :- & causal@p(x,u), causal@p'(u,y) \\ causal@p(x,y) & :- & causal@p(x,v), causal@p''(v,y) \\ causal@p(x,x) & :- & trans@p(x,u,v) \end{array}$$

transTree and placesTree:. We would like to have a copy of the ancestor tree stored locally at a node. This will be used to keep communication local in the $notConf$ relation. We use $transTree(x, w, w', w'')$ to store the relation $trans(w, w', w'')$ locally at node x, where w is an ancestor of node x. Similarly, $placesTree(x, z, z')$ stores a local copy of the relation $places(z, z')$ at node x. For all possible peers $p' \in Neighb(p)$:

$$\begin{array}{l} transTree@p(x,x,u,v) \text{:-} trans@p(x,u,v) \\ transTree@p(x,w,w',w'') \text{:-} trans@p(x,u,v), \\ \quad places@p(u,u'), transTree@p'(u',w,w',w'') \\ placesTree@p(x,u,u') \text{:-} trans@p(x,u,v), \\ \quad places@p'(u,u') \\ placesTree@p(x,z,z') \text{:-} trans@p(x,u,v), \\ \quad places@p(u,u'), placesTree@p'(u',z,z') \end{array}$$

notConf:. Finally, the relation *notConf*(w, x, y) captures absence of conflict ($\neg[x \sharp y]$) between two transition nodes in the unfolding, as observed by node w.

First, we do not want a conflict between any transition and the virtual transition node r:

$$\textit{notConf}@p(w,r,x) \;\; :- \;\; trans(x,y,z), trans(w,y',z')$$

Next, we denote by $Mates(p)$ the set of peers that hold a transition that is the grandparent of a grandchild of some transition at p. For each peer $p' \in Mates(p)$, peer p holds the following rules.

$$\begin{array}{lll} notConf@p(x,z,y) & :- & transTree@p(x,z,u,v), \\ & & placesTree@p(x,u,u'), placesTree@p(x,v,v'), \\ & & notConf@p(x,u',y), notConf@p(x,v',y), \\ & & notCausal@p'(y,u), notCausal@p'(y,v) \\ notConf@p(x,z,y) & :- & transTree@p(x,z,u,v), \\ & & placesTree@p(x,u,u'), placesTree@p(x,v,v'), \\ & & notConf@p(x,u',y), notConf@p(x,v',y), \\ & & causal@p'(y,z) \end{array}$$

We are now ready to state the following theorem that ensures the correctness of the distributed construction of the program. Given a Petri net (N, M), let $Prog(N, M)$ denote the distributed dDatalog program, defined as above, consisting of the set of rules at all peers. Let $\mathcal{N}(N, M)$ consists of the possibly infinite set of domain elements representing the unfolding nodes constructed by the program.

$$\mathcal{N}(N,M) = \{c \mid Prog(N,M) \models trans@p(c,c',c'') \text{ for some } p,c',c''\} \cup \{c \mid Prog(N,M) \models places@p(c,c') \text{ for some } p,c'\}$$

THEOREM 2. *Given a Petri net (N, M), there exists a bijection δ from the nodes of its unfolding $Unfold(N, M) = (\widehat{N}, \rho)$ to $\mathcal{N}(N, M)$ s.t. for every peer name p:*

1. *$c \in \widehat{N}$ is a transition node in p and child of place nodes c_1, c_2 iff $Prog(N, M) \models trans@p(\delta(c), \delta(c_1), \delta(c_2))$.*
2. *$c \in \widehat{N}$ is a place node in p and child of a transition node c_1 in peer p iff $Prog(N, M) \models places@p(\delta(c), \delta(c_1))$*

2'. *$c \in \widehat{N}$ is a place node in p and is a root of the unfolding iff $Prog(N, M) \models places@p(\delta(c), r)$,*

3. *$c \in \widehat{N}$ is a transition mapped by ρ to a Petri net node c' on peer p iff $Prog(N, M) \models map@p(\delta(c), c')$.*

The proof follows from the structure of the rules defining $trans$, $places$ and map, and is based on Lemma 1 that is proved by induction on the depth of the unfolding.

LEMMA 1. *For each transition nodes c (residing in peer p) and c', in* Unfold(N, M)*, we have:*
$\neg[c' \preceq c]$ iff $Prog(N, M) \models notCausal@p(\delta(c), \delta(c'))$;
$\neg[c \sharp c']$ iff $Prog(N, M) \models$ notConf$@p(\delta(c), \delta(c), \delta(c'))$.

We conclude this subsection with two remarks regarding the structure of our program.

Remark 3:. Since there is a one-to-one correspondence between place nodes and the transitions creating them, one can avoid computing and storing them and simply infer the relevant information, when needed, from that of their parents. We have chosen to give the above variant for simplicity of the presentation. The more space conscious variant can be easily inferred.

Remark 4:. The program defines two relations that are complements of each other, $causal$ and $notCausal$. The computation of one could have been saved by using negation. Note that the negation has a stratified flavor: The $notCausal$ relation of two nodes can be determined once the causal relationship for all their ancestors is determined, and cannot be effected by later node creations. Consequently, whenever $notCausal$ is needed in the $trans$ rules for the creation of a new node, it can be inferred from the $causal$ relation among the previously created nodes. Extensions of Magic Sets for Datalog with negation were studied, e.g. in [29, 15] and similar extensions are applicable to the distributed dQSQ setting. In this paper we restricted our attention to positive dDatalog since one of the main goals is to show that even this restrictive setting is powerful enough to model and optimize real life distributed applications.

4.2 Diagnosis of an alarm sequence

Let p_0 be the supervisor site, and A the alarm sequence received by p_0. Each peer p_i provides a description of the transitions in its Petri net, along with the alarm symbols which they emit in the atom $petriNet@p_i(c, a, c', c'')$, which states that transition node c at peer p_i is a child of place nodes c', c'' and emits the alarm a. These base relations describing the Petri net, together with relations $trans@p_i$ describing the unfolding, will be used by the supervisor p_0 to solve the diagnosis problem.

Here again, a crucial point is that p_0 defines its Datalog program *locally*. The rules are based solely on its view of the Petri net, as presented by the alarm sequence it received.

Let k be the number of peer names in the sequence. p_0 first splits the alarm sequence A into k subsequences, one per peer, each being the restriction of A to the alarms emitted by that peer. These subsequences are represented in a base relation called $alarmSeq$ as follows. Consider the subsequence $A_p = (a_1, p), \ldots, (a_n, p)$ of the alarms emitted by a peer p. To encode the index of the alarms in A_p we use $n+1$ distinct constants $c_0 \ldots, c_n$. The relation contains, for $i = 1 \ldots n$, the atom
$alarmSeq(c_{i-1}, a_i, p, c_i)$:-
Additionally it defines the following intensional relations. The first set of rules below constructs, for increasingly larger prefixes of the alarm sequence, configurations that match this prefix. To simplify the presentation, let us first assume that *all the alarms come from a single peer p*. We will explain afterwords how to generalize the given rules for an arbitrary number of peers.

Each (diagnosis) configuration for an alarm sequence prefix of length i is assigned some id. It is obtained from a configuration of length $i-1$ by extending it with one additional transition corresponding to the ith alarm in the sequence. This construction is described by the relation $configPrefixes$. The relation *configPrefixes*$@p_0(id, id', x, i)$ holds iff id is the identifier of prefix configuration of length i, constructed by extending a shorter configuration with id id' with the transition node x. The ids of the prefix configurations are generated using the Skolem function h.

We initialize with sequences of length zero using the constant virtual transition node r, as in rule (††) above:

$$configPrefixes@p_0(h(r), h(r), r, c_0) \text{ :-}$$

Then, we recursively construct the configurations for longer prefixes. More precisely, let p be the name of the peer emitting the alarms in the sequence. (Recall that for now we assume that all alarms come from a single peer). The supervisor defines a rule of the following form.

$$\begin{aligned} configPrefixes@p_0(h(z,x), z, x, i) \text{ :-} \\ petriNet@p(t, a, c, c'), alarmSeq@p_0(i', a, p, i), \\ configPrefixes@p_0(z, w, y, i'), \\ transInConf@p_0(z, u), transInConf@p_0(z, v), \\ notParent@p_0(z, g(u, c)), notParent@p_0(z, g(v, c')), \\ trans@p(x, g(u, c), g(v, c')) \end{aligned}$$

Intuitively, the rule states that a prefix configuration of length i is constructed by picking a shorter configuration z and two transition nodes u, v in z having children place nodes $g(u, c)$, $g(v, c')$ which (1) can together trigger a new transition x with alarm a, and (2) haven't been used already to trigger some other transition in the configuration.

Observe that the rule uses two auxiliary relations, namely *transInConf* and *notParent*. $transInConf@p_0(z, u)$ states that the transition node u participates in the prefix configuration with id z, and is defined as follows. The construction starts from the last

node in the configuration z, and recursively iterates through the shorter prefix configurations from which it was constructed, collecting their transitions:

$transInConf@p_0(z,x)$:-$configPrefixes@p_0(z,w,x,i)$
$transInConf@p_0(z,x)$:-$configPrefixes@p_0(z,w,y,i)$,
$transInConf@p_0(w,x)$
$transInConf@p_0(h(r),r)$:-

The atom $notParent(z,m)$ holds if the unfolding place node m is not a parent of any transition node in the prefix configuration z. It is built monotonously in the style of *notCausal*: For all peer names p in the alarm sequence A, the supervisor defines the following rule:

$notParent@p_0(z,m)$:-$configPrefixes@p_0(z,w,y,i)$,
$trans@p(y,u,v), m \neq u,$
$m \neq v, notParent@p_0(w,m)$
$notParent@p_0(h(r),m)$:-$places@p(m,y)$

Finally, we select the configurations (and their transitions) corresponding to the full alarm sequence. c_n here is the index of the last alarm in the sequence.
$q(z,x)$:- $configPrefixes(z,w,y,c_n), transInConf(z,x)$

Multiple peers. The above rules assume that all the alarms come from a single peer p, and consider incrementally larger prefixes of the alarms from that peer. When the alarm sequence comes from several peers we need to consider incrementally larger prexes of all the subsequences $A_1,\ldots,A_k$. Correspondingly we replace, in the $configPrefixes$ relation, the index attribute i by a k-ary index recording the position in each of the peers subsequences. The peer p_0 now defines $configPrefixes$ rules for all the peers p in the sequence, with an increment to the k-ary index being an increment to any of the subsequence indexes.

The following theorem states the correctness of the above construction. For a Petri net (N,M) and an alarm sequence A, we use $P_A(N,M,A)$ to denote the distributed dDatalog program consisting of the Petri net and supervisor rules. $Conf(N,M,A)$ denotes the set of configurations computed by the program $P_A(N,M,A)$:

$Conf(N,M,A) = \{C(c') \mid$ for some configuration id $c'\}$
where for each c'
$C(c') = \{c \mid P_A(N,M,A) \models transInConf(c',c)\}$

THEOREM 3. *$Conf(N,M,A)$ is precisely the set of all possible configurations of A in $Unfold(N,M)$ (modulo the bijection δ of Theorem 2 between the nodes of the unfolding to the node ids constructed by the program).*

Remark 5:. Here again, the rules that we give above are not the most efficient storage wise. For instance, the ids of nodes in prefix configurations are copied at each step to form a configuration for a longer prefix. This can be saved by using more complex rules that "chase" the nodes in shorter configurations. Note that the configuration (function term) id holds information about the ids of the shorter configurations from which it was constructed, and can be used, via unification, to identify these configurations. We have chosen the above variant for simplicity of presentation. The more space conscious variant is easily inferred.

4.3 Computation with dQSQ

To perform the diagnosis, the supervisor issues the query $q@p_0(?,?)$, which is evaluated with $dQSQ$. Termination of the computation is guaranteed by the following proposition.

PROPOSITION 1. *On input $q@p_0(?,?)$ (and $P_A(N,M,A)$), dQSQ terminates.*

A main concern is the efficiency of the diagnosis process. Previous research aimed at reducing the portions of the unfolding that are constructed during analysis. Similarly, for dDatalog, dQSQ aims at minimizing the quantity of materialized data.

To quantify the success of dQSQ in this context, we have compared it to a dedicated diagnosis algorithm that has been recently proposed in [8]. The algorithm is aimed at solving the diagnosis problem while materializing minimal portions of the unfolding. We sketch the main principals of this algorithm below. For a full description and study see [8].

Given a Petri net (N,M) and an alarm sequence A, the algorithm (i) models A as a linear Petri net formed by a sequence of transitions emitting the alarms in A, (ii) computes the *product* Petri net of (N,M) and A and unfolds it completely. This product unfolding projects to a prefix of $Unfold(N,M)$ containing only the nodes that are "relevant" for the observed alarm sequence. Intuitively, the product of (N,M) and A is a net whose runs satisfy the constraints imposed by both the original Petri net and the alarm sequence. It explains increasing prefixes of the alarm sequence. Starting from the set M of initially marked places on the Petri net and an empty alarm sequence, one adds, to the net constructed for the prefix of length $i-1$, the transition nodes that emit the i^{th} alarm in the sequence and can extend some configuration of length $i-1$ already in the net. When the last alarm symbol is processed, the net contains all the nodes belonging to the possible configurations, and those are extracted bottom up.

Interestingly, the "generic" use of dQSQ achieves *precisely the same reduction* as the above dedicated algorithm, for the portion of the unfolding materialized during the diagnosis process.

To state this more formally, we use the following notations. Given a Petri net (N,M) and an alarm sequence A,
let $\widehat{Unfold}(N,M,A)$ denote the prefix $Unfold(N,M)$ materialized for the alarm sequence A by the algorithm of [8].

Let $\widehat{P_A}(N,M,A)$ denote the rewritten programs generated from $P_A(N,M,A)$ by the dQSQ algorithm. Let $\widehat{trans}$, $\widehat{places}$ and $\widehat{map}$ denote the union of the adorned $trans$, $places$ and map relations in $\widehat{P_A}(N,M,A)$, and $\widehat{\mathcal{N}}(N,M,A)$ the set of unfolding nodes constructed by the program in these relations. We can show the following by induction on the length of the alarm sequence.

THEOREM 4. *There exists a bijection δ from the nodes of $\widehat{Unfold}(N,M,A)$ to $\widehat{\mathcal{N}}(N,M,A)$, satisfying conditions analogous to those of Theorem 2.*

4.4 Extensions

We considered above a basic diagnosis problem where all the alarms emitted by the analyzed peers have been reported to the supervisor. This can be generalized in several ways.

Hidden transitions. The peers may decide to report to the supervisor only part of the alarms, e.g. when some of the transitions have only internal significance (minor alarms).

Alarm patterns. Rather than analyzing one particular alarm sequence, we may seek explanation of a pattern described by some regular language, e.g., $\alpha.\beta^*.\alpha$.

Constraints on the configurations of interest. One may be interested only in sequences of alarms *not* containing some known patterns, and block the unfolding construction upon detection of those patterns.

In all the above problems the structure of the alarm sequences of interest can be easily described by a regular automaton whose allowed transitions can be encoded in the $alarmSeq$ relation. The construction of the possible configurations then follows the same lines as above. One problem introduced here is that the length of the alarm sequence of interest is not bounded. The number of corresponding solutions, as well as the size of the relevant unfolding, may be infinite. While termination is decidable (due to the finite number of simultaneously marked petri nodes), some gadgets to prevent non terminating computations, such as bounding the depth of the unfolding, are desirable.

5. RELATED WORK & CONCLUSION

Petri-net based computations can be modeled and analyzed using several temporal-logic based formalisms [11]; conversely, net unfoldings have been used for model checking temporal logics [14]. The focus of this paper is on a P2P setting where each peer has only a limited view of the system, and on the optimization of computation. We argue that the management of large amounts of data in distributed peers, naturally motivates the use of the deductive database paradigm and can benefit from the large battery of optimization techniques developed for Datalog. We illustrated this thesis here with the diagnosis problem and QSQ, and believe that other optimization techniques can be similarly employed.

The present work falls under the general umbrella of distributed data management [26]. The problem of distributed query processing in a Web context is very active [10]. The architecture stressed here is P2P. P2P information management [1], is becoming popular with systems such as Kazaa [20] and a number of techniques have been developed to support it, e.g., distributed look-up as in [6, 18]. We showed in this paper a connection between an increasingly popular P2P application - the distributed management of telecommunication systems, and a generic Datalog optimization technique, namely QSQ. Interestingly, when adapted to a distributed setting, the extended QSQ achieves an optimization as good as that previously provided by the dedicated diagnosis algorithms. Furthermore, it allows solving efficiently a much larger class of system analysis problems.

The diagnosis of distributed discrete event systems with asynchronous communication is a relatively new research topic. Decentralized diagnosis is analyzed in [21], for the case of synchronous communication; and [23] addresses fault diagnosis for distributed systems modeled by Petri nets, again with synchronous communication. An extension is proposed in [22] to address the effect of (bounded) communication delays in decentralized diagnosis. Difficulties resulting from communications are also investigated in [30]. While [8] provides a detailed and efficient data structure to represent all solutions of the (centralized) diagnosis of concurrent and asynchronous systems by using unfoldings, [9] investigates the high level orchestration of *distributed* diagnosis for concurrent and asynchronous systems.

We adapted here techniques from recursive query processing [3]. The literature on this topic used to be quite prolific and it is difficult to do justice to all who contributed. A nice entry point to the field is [27]. Bottom-up [34] and top-down [28, 7] optimizations have been proposed. Some works considered it in the context of distributed or parallel query processing. E.g., complexity issues are studied in [12]. Perhaps most relevant to this work are [19, 33] that also consider QSQ-like optimizations with data flow evaluations and a termination detection mechanism that resemble ours (not detailed in the present paper). One should also mention many works on the parallel evaluation of transitive closure [4, 17], and of other limited classes of programs [35]. Finally, other optimization techniques have been proposed for distributed/recursive query processing, that are somewhat orthogonal to dQSQ, e.g., semi-join techniques to minimize communications [25], and the exchange of intensional information [2, 32].

As mentioned in the introduction, dQSQ was originally developed to optimize query evaluation for the ActiveXML system, i.e., a system based on the exchange of XML documents with embedded service calls in the P2P setting. We are currently implementing in the ActiveXML system, the algorithms presented here. In the ActiveXML setting, an intensional relation is an active document that is enriched while new facts are deduced. A rule is simulated by a Web service that produces facts. The service calls other relations (local or external) to obtain flows of tuples and produces as well a flow of tuples. Thus the services are continuous and asynchronous.

6. REFERENCES

[1] S. Abiteboul. Managing an XML Warehouse in a P2P Context, 15th International Conference on Advanced Information Systems Engineering, 2003.

[2] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, T. Milo. Dynamic XML Documents with Distribution and Replication. SIGMOD 2003.

[3] S. Abiteboul, R. Hull, V. Vianu. Foundations of Databases. AddisonWesley, 1995.

[4] R. Agrawal, H.V. Jagadish. Multiprocessor Transitive Closure Algorithms. Procs. Intl. Symp. on Database in Parallel and Distributed Systems, 1988

[5] Active XML Web site, http://activexml.net

[6] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. Looking up data in P2P systems. CACM 46(2): 43-48, 2003.

[7] F. Bancilhon, D. Maier,Y. Sagiv, J.D. Ullman. Magic sets and other strange ways to execute logic programs. PODS, 1986.

[8] A. Benveniste, E. Fabre, S. Haar, C. Jard. Diagnosis of asynchronous discrete event systems, a net unfolding approach. IEEE Transactions on Automatic Control 48(5):714–727, May 2003.

[9] A. Benveniste, E. Fabre, S. Haar, C. Jard. Distributed Monitoring of Concurrent and asynchronous Systems. Proc. CONCUR '03, LNCS 2761, pp. 1–26.

[10] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. VLDB Jour., 10:48, 2001.

[11] E.M. Clarke, E.A. Emerson, A.P. Sistla. Automatic verification of finite–state concurrent systems using temporal logic. ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

[12] S.S. Cosmadakis, P.C. Kanellakis. Parallel Evaluation of Recursive Rule Queries. PODS, 1986.

[13] J. Engelfriet. Branching Processes of Petri Nets. Acto In formatica 28, 1991, pp. 575-591.

[14] J. Esparza. Model Checking Using Net Unfoldings. Science of Computer Programming vol 23, pp. 151–195, 1994.

[15] W. Faber, G. Greco, N. Leone. Magic Sets and their Application to Data Integration. To appear in ICDT'05.

[16] J. Hellerstein, B. T. Loo and I. Stoica. Customizable Routing with Declarative Queries, HotNets-III, 2004.

[17] M.A.W. Houtsma, P.M.G. Apers, S. Ceri. Distributed Transitive Closure Computations: The Disconnection Set Approach. VLDB, 1990.

[18] M. Harren, J. Hellerstein, R. Huebsch, B. Thau Loo, S. Shenker, I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks, Peer-to-Peer Systems Int. Workshop, 2002.

[19] G. Hulin. Parallel Processing of Recursive Queries in Distributed Architectures. VLDB, 1989.

[20] Kazaa, www.kazaa.com/

[21] R. Debouk, S. Lafortune, D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete event systems. Disc. Event Dyn. Systems: theory and application vol. 10 no. 1/2, pp 33-86, 2000.

[22] R. Debouk, S. Lafortune, D. Teneketzis. On the effect of communication delays in failure diagnosis of decentralized discrete event systems. Control group rep. CGR00-04, Univ. Mich. Ann Arbor, subm. for publication, 2001.

[23] S. Genc, S. Lafortune. Distributed diagnosis of discrete-event systems using Petri nets. Proc. ICATPN 2003, LNCS 2679, pp 316–336, 2003.

[24] K. McMillan. Using Unfoldings to avoid the state explosion pro blem in the verification of asynchronous circuits. 4th CAV Workshop, pp.164-174, 1992.

[25] W. Nejdl, S. Ceri, G. Wiederhold. Evaluating Recursive Queries in Distributed Databases. TKDE 5(1):104–121, 1993.

[26] M.T. Ozsu, P. Valduriez. Principles of Distributed Database Systems. Prentice Hall, 1991.

[27] Parallelism in Logic Programs, Raghu Ramakrishnan, Proceedings of Symposium on Principles of Programming Languages, 1990

[28] J. Rohmer, R. Lescoeur. La Methode Alexandre une solution pour traiter les axiomes recursifs dons les bases de donnees deductives. Colloque Reconnaissances de Forme et Intelligence Artificielle, Grenoble, 1985

[29] K. Ross. Modular Stratification and Madic Set for DATALOG Programs with Negation.

[30] R. Sengupta. Diagnosis and communications in distributed systems. Proc. WODES 1998 pp. 144-151, IEE, London.

[31] `The SWAN project. http://swan.elibel.tm.fr`

[32] J. Trevor, D. Suciu. Dynamically Distributed Query Evaluation. PODS, 2001.

[33] A. van Gelder. A Message Passing Framework for Logical Query Evaluation. SIGMOD, 1986.

[34] L. Vieille, ”Recursive axioms in deductive databases: The Query-Subquery approach,” in Proc. Int. Conf. Expert Database Syst., L. Kerschberg, Ed., Charleston, 1986.

[35] O. Wolfson. Sharing the Load of Logic-Programming Evaluation. Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, 1988.