DMaC: Distributed Monitoring and Checking

Wenchao Zhou Oleg Sokolsky Boon Thau Loo Insup Lee

Department of Computer and Information Science, University of Pennsylvania,
3330 Walnut Street, Philadelphia, PA 19104-6389
`{wenchaoz,sokolsky,boonloo,lee}@cis.upenn.edu`

Abstract. We consider monitoring and checking formally specified properties in a network. We are addressing the problem of deploying the checkers on different network nodes that provide correct and efficient checking. We present the *DMaC* system that builds upon two bodies of work: the *Monitoring and Checking (MaC)* framework, which provides means to monitor and check running systems against formally specified requirements, and *declarative networking*, a declarative domain-specific approach for specifying and implementing distributed network protocols. *DMaC* uses a declarative networking system for both specifying network protocols and performing checker execution. High-level properties are automatically translated from safety property specifications in the MaC framework into declarative networking queries and integrated into the rest of the network for monitoring the safety properties. We evaluate the flexibility and efficiency of *DMaC* using simple but realistic network protocols and their properties.

1 Introduction

In recent years, we have witnessed a proliferation of new overlay networks that use the existing Internet to enable deployable network evolution and introduce new services. Concurrently, new Internet architectures and policy mechanisms have been proposed to address challenges related to route oscillation and slow convergence of Inter-domain routing.

Within the networking community, there is a growing interest in formal tools and programming frameworks that can facilitate the design, implementation, and verification of new protocols. One of the most commonly proposed approaches is based on *runtime verification* [12, 17], which provides debugging platforms for verifying properties of protocols at runtime. This approach typically works by providing programming hooks that enable developers to check properties of an executing distributed system at runtime.

Existing approaches are often platform dependent and hard to be generalized. The runtime checks are tightly coupled with the implementation and, as a result, cannot be easily reused across different execution environments, or be used to compare different implementations of the same protocol written in different programming languages. Moreover, given that the properties are specified at the implementation level, formal reasoning and analysis are not possible.

To address the above shortcomings, we present *DMaC*, a distributed monitoring and checking platform. *DMaC* builds upon two bodies of work: (1) the *Monitoring and Checking (MaC)* framework [11], which provides means to monitor and check running systems against formally specified requirements, and (2) *declarative networking* [14, 13], a declarative domain-specific approach for specifying and implementing distributed network protocols.

The original MaC framework was designed conceptually as centralized monitoring systems. *DMaC* achieves the distributed capability via the use of *declarative networking*, where network protocols are specified using a declarative logic-based query language called *Network Datalog* (*NDlog*). In prior work, it has been shown that traditional routing protocols can be specified in a few lines of declarative code [14], and complex protocols require orders of magnitude less code [13] compared to traditional imperative implementations. The compact and high-level specification enables rapid prototype development, ease of customization, optimizability, and potentiality for protocol verification.

In *DMaC*, the safety properties of a distributed system are first specified using a platform independent formal specification language called MEDL. These property specifications are then compiled into declarative networking code for execution. Since declarative networks utilize a distributed query engine to execute its protocols, these checks can be expressed as *distributed monitoring queries* in *NDlog*. This paper makes the following three contributions:

- ***DMaC* platform:** The system allows us to specify desired properties of protocols independent of their implementation and abstract away the physical distribution, generate run-time checkers for the properties and deploy them across the network, seamlessly integrated within the *NDlog* engine.
- **Formal specifications to declarative networks:** We show that formal specifications can be automatically compiled to distributed queries. Moreover, given the query-based approach used in declarative networks, we illustrate the potential of applying existing database query optimizations in *DMaC* for efficient plan generation and dynamic reoptimization.
- **Implementation and evaluation:** We have performed evaluation of *DMaC* on several representative examples deployed over a cluster. Results demonstrate feasibility of the approach, in terms of both performance overhead due to monitoring queries, and functionality of the property language.

2 Background

2.1 Declarative Networking

Declarative query languages such as *Network Datalog* (*NDlog*) are a natural and compact way to implement a variety of routing protocols and overlay networks. For example, some traditional routing protocols can be expressed in a few lines of code [14], and the Chord DHT in 47 lines of code [13]. When compiled and executed, these declarative networks perform efficiently relative to imperative implementations, while achieving orders of magnitude reduction in code size.

The compact specifications enable ease of customization and adaptation, where protocols can be adaptively selected and composed based on policies and the changing network environment.

A *NDlog* program is a distributed variant of Datalog which consists of a set of declarative *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and ... and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *literals* that constitute the *body* of the rule. Literals are either *predicates* with *attributes* (which are bound to variables or constants by the query), or boolean expressions that involve function symbols (including arithmetic) applied to attributes. In Datalog, rule predicates can be defined with other predicates in a cyclic fashion to express recursion. The order in which the rules are presented in a program is semantically immaterial; likewise, the order of predicates appearing in a rule is not semantically meaningful. Commas are interpreted as logical conjunctions (`AND`). The names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter.

As a running example throughout the paper, the following four *NDlog* rules implement the *pathVector* protocol that computes the shortest paths between all pairs of nodes in a network.

```
materialize(link,keys(1,2),infinity).
materialize(path,keys(3),infinity).
materialize(bestCost,keys(1,2),infinity).
materialize(route,keys(1,2),infinity).

p1 path(@S,D,P,C) :- link(@S,D,C),P=f_initPath(S,D).
p2 path(@S,D,P1,C1+C2) :- link(@S,Z,C1), route(@Z,D,P,C2),
                          f_memberOf(S,P)=false, P1=f_concat(S,P).
p3 bestCost(@S,D,MIN<C>) :- path(@S,D,C,P).
p4 route(@S,D,P,C) :- path(@S,D,P,C), bestCost(@S,D,C).
```

In *NDlog*, each predicate contains a *location specifier*, which is expressed with `@` symbol followed by an attribute. This attribute is used to denote the source location of each corresponding tuple. For instance, all `link`, `path` and `route` tuples are stored based on the `@S` address field.

The above NDlog program is executed as distributed stream computations in a recursive fashion. Rule `p1` takes `link(@S,D,C)` tuples, and computes all the single-hop paths `path(@S,D,P,C)` where the user-defined function `f_initPath` initializes the path vector as `P=[S,D]`.

In rule `p2`, multi-hop paths are computed. Informally, it means that if there is a link between `S` and `Z` with cost `C1`, and the route between `Z` and `D` is `P` with cost `C2`, then there is a path between `S` and `Z` with cost `C1+C2`. Additional predicates are used for computing the path vector: the built-in function `f_memberOf` is used to drop any paths with loops; if no loop is found, a new path `P1` (from `S` to `D` via intermediate node `Z`) is created by function `f_concat`.

In rule `p3`, the `bestCost` is defined as the minimum cost among all paths. Finally, rule `p4` is a local Datalog rule used to compute the shortest path with the lowest cost based on the local `bestCost` table.

Declarative networking also incorporates the support of *soft-state* derivations which are commonly used in networks. In the soft state storage model, all data (input and derivations) have an explicit "time to live" (TTL) or lifetime, and all tuples must be explicitly reinserted with their latest values and a new TTL or they are deleted. To support this feature, an additional language feature is added to the *NDlog* language, in the form of a `materialize` keyword at the beginning of each *NDlog* program that specifies the TTLs of predicates. For example, the definition `materialize(link,keys(1,2),10)` specifies that the `link` table has its primary key set to the first and second attributes (denoted by `keys(1,2)`)[1], and each `link` tuple has a lifetime of 10 seconds. If the TTL is set to infinity, the predicate will be treated as *hard-state*.

The soft-state storage semantics are as follows. When a tuple is derived, if there exists another tuple with the same primary key but differs on other attributes, an *update* occurs, in which the new tuple replaces the previous one. On the other hand, if the two tuples are identical, a *refresh* occurs, in which the TTL of the existing tuple is extended. For a given predicate, in the absence of any `materialize` declaration, it is treated as an *event* predicate with zero lifetime. Since events are not stored, they are primarily used to trigger rules periodically or in response to network events.

Soft-state tuples are deleted upon expiration. In contrast, hard-state tuples are deleted via *cascaded deletions*. For instance, when existing `links` are deleted, the deletions have to be *cascaded*, resulting in deletions of previously derived `path` tuples, and updates to `bestPath` tuples. The *NDlog* language also generates events when table insertions and deletions occur. For instance, `link_ins` and `link_del` are generated whenever `link` tuples are inserted and deleted respectively.

2.2 Runtime Monitoring and Checking

MaC Overview. Continuous monitoring and verification of the run-time behavior of a system can improve our confidence about the system by ensuring that the current execution is consistent with its requirements at run time [9, 18, 6, 15, 19, 7]. We have developed a Monitoring and Checking (MaC) framework for run-time monitoring of software systems [11], which allows us to specify high-level properties, implement checkers for these properties, extract relevant low-level information from the system execution and abstract these low-level observations to match the level of the property specification.

The MaC framework includes two languages: MEDL and PEDL. The Meta-Event Definition Language (MEDL) is used to express properties. Its formal

[1] Tables are maintained in P2 following the *set* semantic, where primary keys are the unique identifications of the tuples stored in a table. Upon receiving a tuple with the identical primary key as an existing tuple, the table will be updated by replacing the old tuple with the more recent one.

semantics are similar to the semantics of a past-time linear-time temporal logic. It can be used to express a large subset of safety properties of systems, including timing properties. We use events and conditions to capture and reason about temporal behavior and data behavior of the target program execution; events are abstract representations of time progress and conditions are abstract representations of data. Primitive Event Definition Language (PEDL) describes primitive high-level events and conditions used in MEDL properties in terms of system objects. PEDL defines what information is sent from the filter to the event recognizer, and how it is transformed into events used in high-level specification by the event recognizer. Unlike MEDL, PEDL depends on the target system, since it has to refer to observations made directly on the system.

The framework includes two main phases: static phase and dynamic phase. During the static phase, i.e., before a target program is executed, properties are specified in MEDL, along with the respective PEDL mapping. Then, also in the static phase, run-time components such as a filter, an event recognizer, and a run-time checker are generated. During the dynamic phase, the target program, instrumented with the observation filter, is executed while being monitored and checked with respect to the properties. The filter tracks changes of monitored objects and sends observations to the event recognizer. The event recognizer performs the abstraction of low-level observations into high-level events according to the supplied mapping. Recognized events are sent to the run-time checker. Although the event recognizer can be incorporated into the checker, we separate them to clearly distinguish high-level properties, independent of a particular implementation, from low-level information extraction, which by necessity applies to a given implementation. A run-time checker determines whether or not the current execution history satisfies the properties. Checking is performed incrementally, without storing unnecessary history information.

We have implemented a prototype of the MaC for Java programs, called Java-MaC [10]. Java-MaC targets Java bytecode, providing automatic instrumentation of the target program and automatic generation of the checker, event recognizer, and filter. Although the existing implementation can be easily extended to monitor other kinds of target systems by providing a different filter, it is difficult to apply it to distributed systems.

MEDL Specification Language. The language MEDL is built around a two-sorted logic that describes events and conditions. In this paper, we use a parametric variant of MEDL described in [23], but without explicit quantification. We present MEDL syntax and semantics using examples. For the formal presentation, we refer the reader to [10, 23].

In MEDL, events are instantaneous observations. Primitive events are received from the monitored system, while composite events are derived in the checker. Events can carry attributes that give additional information about the occurrence. For example, event $route(s, d, p)$ is observed when the node s enters the route to the node d into its routing table, and the route follows the path p, and these three values are supplied as attributes of the event. In addition, each event has the timestamp of its occurrence, interpreted by the checker clock. Unlike events, conditions have durations: once a condition becomes true, it is true continuously until it becomes false. Primitive conditions are defined within the monitored system, which notifies the checker about changes to the condition value. Conditions can be parametrized by attributes of events that are used in the definition of a condition, as described below.

Primitive events are defined using the *import declaration*: *import event* $e(\bar{X})$ specifies that event e can observed directly from the monitored system and that every occurrence of e carries attributes $\bar{X} = (x_1, \ldots, x_k)$. A conjunction of two events $e(\bar{X}) = e_1(\bar{X_1}) \wedge e_2(\bar{X_2})$ is an event that occurs when both e_1 and e_2 occur simultaneously. We require that $var(\bar{X}) \subseteq var(\bar{X_1}) \cup var(\bar{X_1})^2$; that is, attributes of the composite event can come only from the events and conditions used in its definition. For example, event $e(x, y) = e_1(p, x) \wedge e_2(x, y, 2)$ occurs at time t only if both e_1 and e_2 occur at time t, such that the last attribute of e_2 is the constant 2 and the common attribute x has the same value in both occurrences. Disjunction of events is defined similarly.

Given a condition c, event $(e \,\mathrm{when}\, c)$ occurs if e occurs while the condition c is true. As an example, consider $e_c(x_1, x_2) = e(x_1) \,\mathrm{when}\, c[x_1, x_2]$. Let conditions $c[1, 1]$, $c[1, 2]$, and $c[2, 1]$ be true and $c[1, 3]$ be false. When event $e(1)$ occurs, it will cause simultaneous occurrences of events $e_c(1, 1)$ and $e_c(1, 2)$.

Each condition $c[\bar{X}]$ is associated with two events, start_c and end_c, both of which have $\bar{X}$ as attributes. Event start_c occurs when c becomes true, while end_c occurs with it becoming false. A primitive condition $c[x_1, \ldots, x_n]$ represents the directly observed state of the monitored system. Whenever a ground instance $c(p_1, \ldots, p_n)$ changes its value, the checker receives an occurrence of $\mathrm{start}_c(p_1, \ldots, p_n)$ or $\mathrm{end}_c(p_1, \ldots, p_n)$, corresponding to the new value of the ground condition, from the event recognizer.

Conjunction, disjunction, and negation of conditions are defined in the expected way. Any two events $e_1(\bar{X_1})$, $e_1(\bar{X_1})$ define a condition $[e_1, e_2)$, which is parametrized with $var(\bar{X_1}) \cup var(\bar{X_2})$. This condition is true if there has been an occurrence of e_1 in the past and, since then, there has been no occurrence of e_2 with matching attributes. A time-bounded variant of this expression, $[e_1, e_2)_{<t}$, means that the condition becomes false t time units after the occurrence of e_1 if e_2 does not arrive within these t time units.

MEDL also uses auxiliary variables that allow us to store additional history information. Predicates over auxiliary variables can be used to define new conditions. Semantically, auxiliary variables are similar to conditions, except that a ground instance of an auxiliary variable evaluates to a numerical rather than boolean value. Auxiliary variables are updated in response to events in guarded commands of the kind $e(\bar{X}_e) \rightarrow \{v[\bar{X}_v] := expr(\bar{X}_e)\}$, where $var(\bar{X}_v) \subseteq var(\bar{X}_e)$ and *expr* is an arithmetic expression over $\bar{X}_e$.

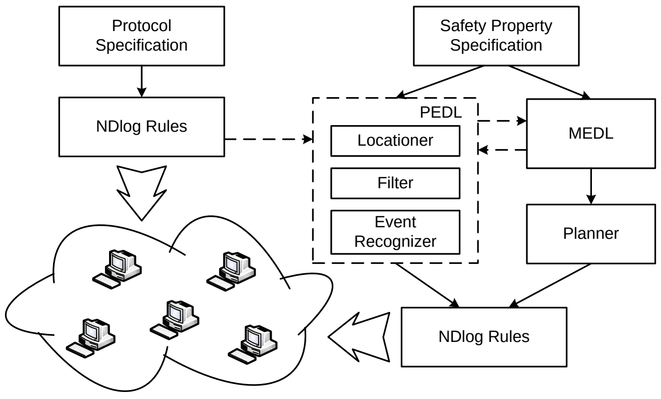


Fig. 1. Architectural Overview of *DMaC*

3 *DMaC* Architecture

Figure 1 shows the architecture of *DMaC*. In the above figure, the monitored network is specified using *NDlog* as a collection of rules. The system can be used for monitoring existing legacy networks (where the network state is exposed as distributed tables that one can query over), or declarative networks themselves. The latter provides more opportunity for fine-grained analysis, since the protocol and monitoring queries themselves are based on the same declarative query language.

Separately, safety properties of the protocols are described in MEDL. The MEDL specifications are location-agnostic. PEDL scripts are then used to specify the distributed nature of the monitored property, in the form of an *event recognizer* that takes as input the events that arrive at a node, and a *locationer* that determines the locations where generated events from MEDL are to be sent. MEDL properties are translated into *NDlog* rules without location information. The planner module then utilizes the PEDL specifications in combination with query optimization techniques to decide on the allocation of the generated rules and annotate the *NDlog* rules with location information. The generated *NDlog* rules are then deployed in the network as monitoring queries.

3.1 Example Network Property: Route Persistence

To illustrate the use of *DMaC*, we consider the *route persistence* property [16], which tracks the duration that each computed route persists without changing. We will introduce more complex examples in Section 3.3.

In the normal situation, we expect routes, once established, to be stable for some time before changing again. Therefore, upon each route update message,

[2] $var(\bar{X})$ are the variables in $\bar{X}$

we raise *persistenceAlarm* when changes occur too quickly (e.g., less than 10 seconds) after the previous update. In the context of deployed network protocols, rapid changes to computed routes could be a symptom of more serious issues, such as the lack of convergence in the protocol.

$$
\begin{array}{lll}
\textit{import} & \textit{event} & \quad \textit{rtMsg}(s,d,p) \\
\textit{var} & \textit{routeStored}[s,d], \textit{timeStored}[s,d] & \\
\textit{event} & \textit{newRoute}(s,d,p) & = \textit{rtMsg}(s,d,p) \text{ when } \textit{routeStored}[s,d] \neq p \\
\textit{event} & \textit{persistenceAlarm}(s,d) & = \textit{newRoute}(s,d,p) \text{ when} \\
 & & \quad \textit{time}(\textit{newRoute}(s,d,p)) - \textit{timeStored}[s,d] < 10 \\
 & \textit{newRoute}(s,d,p) & \rightarrow \{\textit{routeStored}'[s,d] := p; \\
 & & \quad \textit{timeStored}'[s,d] := \textit{time}(\textit{newRoute}(s,d,p); \}
\end{array}
$$

The above MEDL program uses the `time(newRoute)` construct to denote the time of the `newRoute` event. All events have an associated timestamp. To avoid clock synchronization issues, *DMaC* takes the convention that events are assigned a receiver-based timestamp, and this timestamp is retrievable via the `time` construct above. The translation of MEDL into *NDlog* is as follows:

```
sm1 newRoute(@S,D,P,T) :- rtMsg(@S,D,P), routeStored(@S,D,P1),
                          P1!=P, T:=f_now().
sm2 routeStored(@S,D,P) :- newRoute(@S,D,P,T).
sm3 timeStored(@S,D,T) :- newRoute(@S,D,P,T).
sm4 persistenceAlarm(@S,D) :- timeStored(@S,D,T1),
                              newRoute(@S,D,P,T), T-T1<10.
```

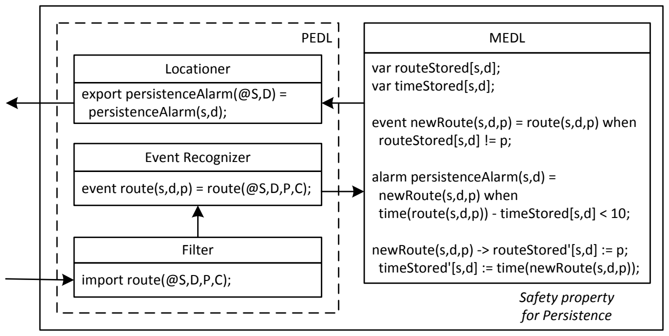
In the above program, `f_now` is a user-defined function that returns the current local time of the node where the rule is triggered. In order to check the property over the path vector protocol, we need to connect the two sets of *NDlog* rules, specifying how the imported event *rtMsg* is produced by the protocol specification. For this, we add the following PEDL statement:

```
export event rtMsg: rtMsg(@S,D,P) :- route(@S,D,P,C).
```

This statement serves several purposes. First, it specifies that *rtMsg* originates from changes to the `route` table of node `S` in the protocol specification and, second, the *rtMsg* event is raised at the same node, `S`, where the route table is stored. Finally, it projects away irrelevant attributes of the table, in this case, `C`. PEDL rules can also be used to insert additional attributes into the high-level event. For example, if `rtMsg` was to be processed at a node different from `S`, we may choose to insert the origination timestamp as an additional attribute of `rtMsg` in order to support the reasoning about propagation delay in property specifications.

The PEDL script also contains another statement:

```
export persistenceAlarm(@S,D)
```



```
export persistenceAlarm(@S,D) =
  persistenceAlarm(s,d);
```

```
event route(s,d,p) = route(@S,D,P,C);
```

```
import route(@S,D,P,C);
```

```
var routeStored[s,d];
var timeStored[s,d];

event newRoute(s,d,p) = route(s,d,p) when
  routeStored[s,d] != p;

alarm persistenceAlarm(s,d) =
  newRoute(s,d,p) when
  time(route(s,d,p)) - timeStored[s,d] < 10;

newRoute(s,d,p) -> routeStored'[s,d] := p;
  timeStored'[s,d] := time(newRoute(s,d,p));
```

Fig. 2. MEDL and PEDL specifications for the Persistence Alarm

which specifies that *persistenceAlarm* event is exported - that is, it can be used in other MEDL properties and also in other network protocols. For exported events, we specify the location where the event is raised.

From the locations assigned to the imported and exported events, the planner module decides on the deployment of MEDL rules to network nodes. In this simple example, where both exported and imported events are local to the node `S`, it is clear that the checker should also be located at `S`. In more complicated cases, intermediate results of MEDL evaluation may be placed at different nodes. We use known techniques from query optimization to compute this placement. More details are given in Section 4.3.

3.2 MEDL Extensions for Sliding Window Event Correlation

In practice, several monitoring scenarios in the networking domain require the ability to reason about event occurrences that arrive within a given time window. We provide an example in Section 3.3 but briefly outline language extensions to MEDL in order to support such correlation:

First, to make reasoning about event correlation easier, we also define a time-bounded conjunction of events $e_1 \wedge_{<t} e_2$ and $e_1 \wedge_{\leq t} e_2$. This event occurs if occurrences of e_1 and e_2 are separated by less (respectively, no more) than t time units. Note that $e_1 \wedge_{\leq t} e_2$ can be expressed as $e_2 \wedge \text{end}([e_1, e_2)_{<t}) \vee e_1 \wedge \text{end}([e_2, e_1)_{<t})$.

Second, we define time-based and count-based sliding windows for a given event $e(\bar{X})$. Window $[e(\bar{X})]_{\leq t}$ contains all event occurrences of e such that for any two occurrences in the window, the difference between their timestamps does not exceed t. A similar definition is made for strict inequality. Window $[e(\bar{X})]_{\#n}$ contains n last occurrences of e, if the execution trace has more than n occurrences, and all occurrences of e otherwise. The role of sliding windows is to allow us to use aggregations of event attributes in the definition of events and conditions. We use functions such as minimum, maximum, sum, average, etc. to define aggregations. For example, *condition* $c[x] = \max_p[e(x,p)]_{\leq 5} > 10$ asserts the maximum value of the attribute p in the events that occurred in the last 5 time units should be larger than 10. Note that this expression effectively defines a separate sliding window for each value of x encountered so far.

3.3 Other Examples

We consider additional network properties to illustrate the flexibility of *DMaC*, as well as demonstrate intuitively the translation of MEDL into *NDlog*. The examples are representative of typical queries that involve aggregation and event correlation. Other examples unrelated to network routing (e.g. ensuring the correctness of distributed hash table routing, multicast trees, etc) are also possible within our framework.

Data Plane Monitoring. In the previous example, we considered a property that, when violated, indicates a problem with the *control plane* of the network. Here, we consider a property that indicates a problem with the *data plane*, that is, the transmission of packets over established routes. Often, sudden changes in the flow rate of data across the network indicate a problem with the network. We define flow rate as the average number of packages that are transmitted in a period of time (for example, during the last 60 seconds). We raise `flowAlarm` when the flow rate between two nodes exceeds a threshold (say 100KB). In order to calculate flow rate, we use the new sliding window feature of MEDL.

$$\begin{array}{lll} \textit{import event} & & \textit{package}(s,d,\textit{size}) \\ \textit{event} & \textit{rateAlarm}(s,d) = & \textit{start}(\sum_{\textit{size}}[\textit{package}(s,d,\textit{size})]_{60} > 100000) \end{array}$$

As the result of the translation to *NDlog*, we obtain the following rules:

```
materialize(package,keys(1,2),60).
ct1 packageSum(@S,D,SUM<Size>) :- package(@S,D,Size).
ct2 rateAlarm(@S,D) :- packageSum_ins(@S,D,Sum), Sum>100000.
```

Distributed Time-based Event Correlation. Often, it is necessary to correlate events that are raised by different nodes in the network. In particular, simultaneous problems in the control and data planes of the network often indicate an attack on the network. In this example, we consider a set of gateways, each of which is responsible for the health of a set of sensor nodes. The gateways need to correlate persistence (`pAlarms`) and rate alarms (`rAlarms`) sent from different nodes. We revisit this example in Section 4.3 to motivate our approach to location assignment for MEDL rules. The property utilizes the node-to-gateway mapping as an imported condition, allowing the mapping to change dynamically.

$$\begin{array}{lll} \textit{import} & \textit{condition} & \quad\quad\;\; \textit{gateway}(s,m) \\ \textit{event} & \textit{gatewayAlarm}(m,z) & = \textit{persistenceAlarm}(s,z) \wedge_{<5} \textit{rateAlarm}(z,d) \\ & & \quad \text{when } \textit{gateway}(s,m) \\ \textit{alarm} & \textit{attackAlarm}(m) & = \textit{start}(\#[\textit{gatewayAlarm}(m,z)]_{3600} > 4) \end{array}$$

The translation to *NDlog* is as follows:

```
materialize(gateway,keys(1),infinity).
materialize(pAlarms,keys(1,2),5).
materialize(rAlarms,keys(1,2),5).
materialize(gatewayAlarm,keys(1,2),3600).

cm1 pAlarms(@M,S,D) :- persistenceAlarm(@S,D,Stat), gateway(@S,M).
cm2 rAlarms(@M,S,D) :- flowAlarm(@S,D,Stat), gateway(@S,M).
cm3 gatewayAlarm(@M,Z) :- pAlarms(@M,S,Z), rAlarms(@M,Z,D).
cm4 alarmNum(@M,COUNT<>) :- gatewayAlarm(@M,Z).
cm5 attackAlarm(@M,Count) :- alarmNum_ins(@M,Count), Count>4.
```

4 Translating MEDL into *NDlog*

In this section, we present a general algorithm that translates MEDL rules into the corresponding *NDlog* programs. Figure 3 shows the steps involved in translation: *MEDL normalization*, *Datalog generation*, and *Optimized* NDlog *generation*. We focus on the first two steps, and defer the discussion of the third step to Section 4.3.

4.1 MEDL Normalization

In the first step, each MEDL rule is rewritten into a *normalized* MEDL expression, in which each event and condition is defined by an application of exactly one operator, applied to either events or conditions or constants. For example, $e(x,y) = (e_1(x,z) \wedge_{\leq t} e_2(y,z))$ when $(c_1[x,y] \wedge c_2[x,z])$ would be represented as the following three rules:

$$\begin{aligned} e(x,y) &= e_{12}(x,y,z) \text{ when } c_{12}[x,y,z] \\ e_{12}(x,y,z) &= e_1(x,z) \wedge_{\leq t} e_2(y,z) \\ c_{12}[x,y,z] &= c_1[x,y] \wedge c_2[x,z] \end{aligned}$$

We also require that each guarded command updates exactly one variable, which can be achieved by splitting the update block into individual statements and creating a separate guarded command for each.

As the basic components in MEDL, events, conditions and auxiliary variables are translated to tuples in NDlog. An event $e(\bar{X})$ is translated to the tuple `e(X1,...,Xn)`, and the presence of a tuple indicates that the event has occurred. Translation of a condition $c[\bar{X}]$ is `c(X1,...,Xn)`, where the presence of the tuple indicates that the condition is true and the absence of it means that it is false. An auxiliary variable $v[\bar{X}]$ is translated to the tuple `v(X1,...,Xn,V)`, with the last variable storing the current value of the variable. Tuples that correspond to conditions and variables are materialized using `X1,...,Xn` as keys. Events, on the other hand, may carry attributes that need not be used as keys. We use a simple technique to identify such attributes.

An event attribute that is used in the definition of a condition or variable as its parameter is always a part of the key for the event relation. This is to ensure that the event is present when the rule that updates the condition or variable is evaluated. However, if an attribute of an event is used only to update values of auxiliary variables, then attribute need not be a part of the key. Consider event *newRoute* in Figure 2. Its attribute p is not a part of the key, since the event is used in the definition of *persistenceAlarm* and also to update values of variables *routeStored* and *timeStored*, none of which is parameterized by p.

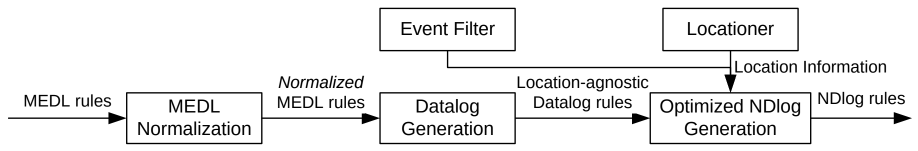


Fig. 3. Translation process from MEDL rules to NDlog rules

4.2 Datalog Generation

The Datalog generation process rewrites normalized MEDL rules into location-agnostic Datalog rules. Figure 4 summarizes the translation algorithm by listing each MEDL rule type and the corresponding Datalog rules.

We categorize normalized MEDL rules into ten different types, of which six are for event generation, three for condition generation and one for variable updates. Due to space constraints, we highlight two particularly interesting translation.

The 3rd row shows a MEDL rule for sliding window based event correlation (see Section 3.2) and the corresponding Datalog rules. The translation result in 4 Datalog rules that used to store the events `e1` and `e2` as soft-state tables e'_1 and e'_2 respectively for a specified lifetime determined by the sliding window size of t seconds. The soft-state tables are then used for correlating the events over the time-interval of t seconds.

In the 8th row, the condition predicate $c(X_1, ..., X_n)$ over auxiliary variables $v_1[\bar{X_1}], ..., v_n[\bar{X_n}]$ is handled by introducing the function predicate `pred` into the

MEDL Rules	Corresponding Datalog Rules
$e(\bar{X}) = e_1(\bar{X_2}) \vee e_2(\bar{X_2})$	$e(X_1, ..., X_n) : -e1(X_{1,1}, ..., X_{1,k}).$ $e(X_1, ..., X_n) : -e2(X_{2,1}, ..., X_{2,m}).$
$e(\bar{X}) = e_1(\bar{X_1})$ when $c[\bar{X_2}]$	$e(X_1, ..., X_n) = e_1(X_{1,1}, ..., X_{1,k}), c(X_{2,1}, ..., X_{2,m}).$
$e(\bar{X}) = e_1(\bar{X_1}) \wedge_{\leq t} e_2(\bar{X_2})$	$materialize(e'_1, keys(1, 2), t).$ $materialize(e'_2, keys(1, 2), t).$ $e'_1(X_{1,1}, ..., X_{1,k}) : -e_1(X_{1,1}, ..., X_{1,k}).$ $e'_2(X_{2,1}, ..., X_{2,m}) : -e_2(X_{2,1}, ..., X_{2,m}).$ $e(X_1, ..., X_n) = e_1(X_{1,1}, ..., X_{1,k}), e'_2(X_{2,1}, ..., X_{2,m}).$ $e(X_1, ..., X_n) = e_2(X_{2,1}, ..., X_{2,m}), e'_1(X_{1,1}, ..., X_{1,k}).$
$e(\bar{X}) = start(c[\bar{Y}])$	$e(X_1, ..., X_n) : -c\_ins(Y_1, ..., Y_m).$
$e(\bar{X}) = end(c[\bar{Y}])$	$e(X_1, ..., X_n) : -c\_del(Y_1, ..., Y_m).$
$c[\bar{X}] = c_1[\bar{X_1}] \wedge c_2[\bar{X_2}]$	$c(X_1, ..., X_n) : -c_1(X_{1,1}, ..., X_{1,k}), c_2(X_{2,1}, ..., X_{2,m}).$
$c[\bar{X}] = c_1[\bar{X_1}] \vee c_2[\bar{X_2}]$	$c(X_1, ..., X_n) : -c_1(X_{1,1}, ..., X_{1,k}).$ $c(X_1, ..., X_n) : -c_2(X_{2,1}, ..., X_{2,m}).$
$c[\bar{X}] = pred(v_1[\bar{Z_1}], ...v_p[\bar{Z_p}])$	$c(X_1, ..., X_n) : -v_1(Z_{1,1}, ..., Z_{1,m_1}, Val_1), ...,$ $v_p(Z_{p,1}, ..., Z_{p,m_p}, Val_p), pred(Val_1, ...Val_p).$
$c[\bar{X}] = [e_1(\bar{X_1}), e_2(\bar{X_2}))$	$c(X_1, ..., X_n) : -e_1(X_{1,1}, ..., X_{1,k}).$ $delete\ c(X_1, ..., X_n) : -e_2(X_{2,1}, ..., X_{2,m}), c(X_1, ..., X_2).$
$e(\bar{X}) \rightarrow \{v[\bar{Z}] := expr(v_1[\bar{Z_1}], ..., v_p[\bar{Z_p}])\}$	$v(Z_1, ..., Z_n, Val) : -v_1(Z_{1,1}, ..., Z_{1,m_p}, Val_1), ...,$ $v_p(Z_{p,1}, ..., Z_{p,m_p}, Val_p), Val := expr(Val_1, ...Val_p).$

Fig. 4. MEDL and corresponding Datalog rules

Datalog rule. The rule is triggered by update events of each variable. As an example, the condition $c[x, y] = v_1[x] + v_2[y] > 5$ will be translated to the corresponding Datalog rule $c(X, Y) : -v_1(X, Val_1), v_2(Y, Val_2), Val_1+Val_2 > 5$.

Note that, the $time(event)$ semantic in MEDL requires recoding the generation timestamp as additional information for an event. If the timestamp of an event is used in a MEDL program, variable T is added to the relation, and $T := f\_now()$ is added to the rule that produces the event. This results in the use of receiver-based timestamps, where each event is timestamped based on the recipient node's time.

4.3 *NDlog* Program Generation and Optimization

The `Filter` and `Locationer` modules in PEDL explicitly indicate the physical locations of the import and export events and conditions used in *DMaC* rules. To deploy *DMaC* programs for distributed monitoring, one needs to further assign the locations where the computations in MEDL (e.g. correlation of events and conditions) should take place. In several instances, different location assignments may result in varying communication overhead, and the specific data placement strategy is determined by factors such as inter-node latency, bandwidth, and also the rate at which events are generated. Interestingly, given our use of declarative networking, the optimization decisions map naturally into *distributed query optimizations*. Our goal in this section is to highlight some of the challenges and opportunities in this space, and set the stage for richer explorations in future.

Motivating Example. We consider a three-node network consisting of nodes n_1, n_2 and n_3. The three events in the network that follow the MEDL rule are:

$e_3(\bar{X_3}) = e_1(\bar{X_1}) \wedge_{\leq t} e_2(\bar{X_2})$, where $e_1(\bar{X_1})$ is located at n_1; $e_2(\bar{X_2})$ is located at n_2; and $e_3(\bar{X_3})$, as the correlation results of e_1 and e_2, is located at n_3. These three events respectively refer to the `persistenceAlarm`, `flowAlarm`, and `attackAlarm` events from the event correlation example in Section 3.3.

According to the locations where the correlation is performed, the compilation of the MEDL rule may result in different sets of NDlog rules, each of which potentially has a distinct execution overhead. There are three potential execution plans for the above MEDL rule:

- **Plan a (correlation at n_1):** Node n_2 sends $e_2(\bar{X_2})$ to n_1. Meanwhile n_1 performs correlation of the received $e_2(\bar{X_2})$ events and the local $e_1(\bar{X_1})$ events, and it sends the resulting $e_3(\bar{X_3})$ to n_3;
- **Plan b (correlation at n_2):** Node n_1 sends $e_1(\bar{X_1})$ to n_2, and n_2 performs the correlation and sends the resulting $e_3(\bar{X_3})$ to n_3;
- **Plan c (correlation at n_3):** Node n_1 and n_2 send the generated events $e_1(\bar{X_1})$ and $e_2(\bar{X_2})$ to n_3, where the correlation is performed.

Once the correlation location is decided, a MEDL rule will be translated into NDlog rules automatically by the compilation process. For instance, taking node n_3 as the correlation location, the above MEDL rule is translated into the following NDlog rules:

```
op1 e1'(@n3,X1) :- e1(@n1,X1).
op2 e2'(@n3,X2) :- e2(@n2,X2).
op3 e3(@n3,X3) :- e1'(@n3,X1), e2'(@n3,X2).
```

Rules `op1` and `op2` ship $e_1(\bar{X_1})$ and $e_2(\bar{X_2})$ from their original location to node n_3, which is the correlation location. Rule `op3` then performs the correlation of these two events and generates $e_3(\bar{X_3})$. Following a similar approach, one can easily write corresponding NDlog rules if the correlation happens at n_1 or n_2.

Cost-based Query Optimizations. When a MEDL rule that involves multiple events and conditions is compiled to *NDlog* implementation, we choose as default to send the contributing events and conditions to the location where the computation result is located (i.e. plan c in the above example).

Plan c may in fact turn out to be an inefficient strategy. Revisiting the correlation example in Section 3.3, if node `n3` is connected via a high-latency, low-bandwidth link to nodes `n1` and `n2`, and the rate at which events `e1` and `e2` are generated is extremely high, this plan would result in overwhelming node `n3`. A superior strategy in this case could be for events to be correlated locally between `n1` and `n2` first. The savings can be tremendous, particularly if these two nodes are connected via a high-speed network, and the actual correlation of `e3` events is infrequent.

Interestingly, one can model the above tradeoffs via *cost-based query optimizations*, where a cost is assigned to each plan, and the plan with the lowest cost is selected. As an example, the bandwidth utilization of the above plans

can be estimated from the rate at which events arrive, and the selectivity of the correlation:
Plan a: $|e_2| * s_{e_2} + r_{e_1,e_2} * |e_1| * |e_2| * s_{e_3}$
Plan b: $|e_1| * s_{e_1} + r_{e_1,e_2} * |e_1| * |e_2| * s_{e_3}$
Plan c: $|e_1| * s_{e_1} + |e_2| * s_{e_2}$
where $|e_i|$ represents the estimated generation rate of event $e_i(\bar{X}_i)$, s_{e_i} represents the message size of $e_i(\bar{X}_i)$, r_{e_1,e_2} is the selectivity of the correlation, i.e. the likelihood that $e_1(\bar{X}_1)$ and $e_2(\bar{X}_2)$ are correlated.

Given a MEDL rule, exhaustive enumeration is required to search for all potential execution plans and select the optimal plan given the cost estimates. In practice, finding the optimal plan is NP-hard. To find approximate solutions that provide efficient plans, commercial database engines utilize dynamic programming [20] in combination with other plan selection heuristics, enabling the query engine to generate efficient plans at reasonable overhead.

In our distributed setting, the optimal plan may change over time as the rate of events and network conditions vary during the monitoring period. In recent database literature, there has been significant interests and progress in the area of *adaptive query optimization* [5] techniques, commonly used to *reoptimize* query plans during execution time. Such runtime reoptimizations are particularly useful in the areas of wide-area data integration and distributed processing of sensor feeds. We intend to explore the use of adaptive query optimizations in *DMaC* as future work.

5 Evaluation

In this section, we perform an evaluation of the *DMaC* system. The goals of our evaluation are two-fold: (a) to experimentally validate the correctness of the *DMaC* implementation, and (b) to study the additional overhead incurred by monitoring rules. Our experiments are executed within a local cluster with 15 quad-core machines with Intel Xeon 2.33GHz CPUs and 4GB RAM running Fedora Core 6 with kernel version 2.6.20, which are interconnected by high-speed Gigabit Ethernet. Our evaluation is based on the P2 declarative networking system [1]. In the experiments, we deploy up to a network size of 120 nodes (eight nodes per physical machine).

For the workload, we utilize the path vector protocol which computes the shortest paths between all pairs of nodes. To evaluate the additional overhead incurred by monitoring safety properties, we deploy two versions of the path vector query, i.e. *PV* and *PV-DMaC*, where *PV* executes pure path vector query presented in Section 2.1, *PV-DMaC* additionally executes the *DMaC* rules presented in Section 3.1 to monitor the route persistence property. When a violation of the property is detected, the specified MEDL/PEDL results in the generation of `persistenceAlarm` events which are exported to a centralized monitor to log all such violations across the network.

As input, each node takes a `link` table pre-initialized with 6 neighbors. After the path vector query computation reaches a fixpoint, i.e. the computation of

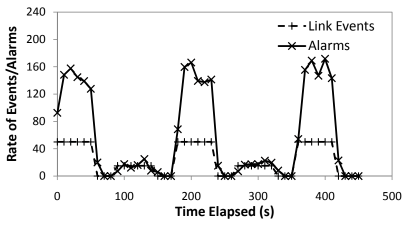


Fig. 5. Number of updates and persistence alarms over time

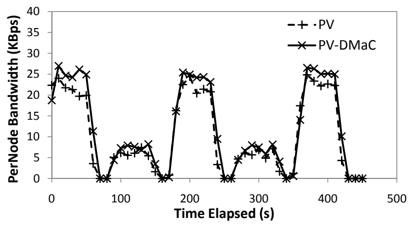


Fig. 6. Per-Node bandwidth (KBps) for monitoring route persistence

all-pairs best paths have completed, we periodically inject network events to induce *churn* (changes to the network topology). To evaluate the performance overhead at high and low rates of network churn, at each 60-second interval, we interleave high churn of 50 link updates per second followed by low churn of 15 link updates per second. As links are added and deleted, the path vector query will recompute the routing tables incrementally.

Figure 5 shows the number of `persistenceAlarms` that are generated per second in response to the link updates. We observe that there is a clear correlation between the rate of link events and alarms: when the network is less stable (i.e. in high churn, such as 0-60 seconds), the persistence property is more likely to be violated due to frequent route recomputations, hence resulting in a higher rate of the `persistenceAlarms`; whereas the rate of the alarms drops significantly when the network is under low churn.

Figure 6 shows the per-node bandwidth overhead incurred by *PV* and *PV-DMaC* as the protocol incrementally recomputes in response to link updates. We observe that *PV-DMaC* incurs only an additional overhead of 11% in bandwidth utilization. The overhead is attributed primarily to the generation of `persistenceAlarms` which are sent to the centralized monitor. We note that in absolute terms, the increase in bandwidth utilization is 2.5KBps, which is well-within the capacity of typical broadband connections.

6 Related Work

Literature on run-time monitoring and checking for networked systems can be divided into two broad categories. One category of papers addresses general-purpose run-time verification problems. These papers are typically concerned with increasing expressiveness of property specification languages and developing time- and space-efficient online checking algorithms. Multiple run-time verification systems [7, 21, 3, 4] reported in the literature are primarily oriented toward monitoring of software. While some of them are capable of collecting observations from multiple nodes in a distributed system [2], they typically assume a centralized checker.

The other category of related work contains papers originating in the networking community. Here, distribution is critical in both collection of observations and in checking. This category of work typically uses simple invariant properties and is concerned with minimization of network traffic. An important point of comparison is [8], which offers a system H-SEND for invariant monitoring of wireless networks.

Our work differs from both of these categories. On one hand, we are using a richer language than the one typically used in correctness monitoring of networks. On the other hand, we address distributed deployment of checkers in the network, the aspect typically not considered in run-time verification literature. Finally, deployment of checkers is tightly integrated with the network deployment itself through *NDlog*, which is a unique feature of our approach.

7 Conclusion and Future Work

We have presented a way to integrate a run-time verification framework into a declarative networking system that is based on the language *NDlog*. The integration allows us to specify high-level, implementation-independent properties of network protocols and applications in the language MEDL, generate checkers for these properties, and deploy the checkers in a distributed fashion across the network. Checkers are generated by translating MEDL properties into *NDlog* and are executed as distributed queries along with the protocol implementations. We use distributed query optimization techniques to derive allocation of checkers to network nodes.

In the future work, we will work to remove restrictions on MEDL constructs introduced in this paper. The restrictions stem from the treatment of event timestamps in a distributed system. Currently, timestamps of events transmitted across the network are assigned based on the local clock of the receiving node, and sender's timestamps may be captured as event attributes. While this is adequate for many of commonly used network properties, a more general treatment is desirable. A possible approach is to introduce the knowledge about the physical distribution of events through the network and extending the notion of a timestamp along the lines of [22].

8 Acknowledgments

This work is based on work supported in part by ONR MURI N00014-07-1-0907, NSF CNS-0721845 and NSF IIS-0812270.

References

1. P2: Declarative Networking. http://p2.cs.berkeley.edu.
2. A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, volume 4337 of *LNCS*, Dec. 2006.
3. F. Chen and G. Rosu. MOP: An efficient and generic runtime verification framework. In *Proceedings of OOPSLA'07*, pages 569–588, 2007.

4. C. Colombo, G. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '08)*, Sept. 2008.
5. A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
6. M. Diaz, G. Juanole, and J.-P. Courtiat. Observer - a concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering*, 20(12):900–913, Dec. 1994.
7. K. Havelund and G. Rosu. Monitoring Java programs with JavaPathExplorer. In *Proceedings of the Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2001.
8. D. Herbert, V. Sundaram, Y.-H. Lu, S. Bagchi, and Z. Li. Adaptive correctness monitoring for wireless sensor networks using hierarchical distributed run-time invariant checking. *ACM Transactions on Autonomous and Adaptive Systems*, 2(3), 2007.
9. F. Jahanian and A. Goyal. A formalism for monitoring real-time constraints at run-time. *20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, pages 148–55, 1990.
10. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance approach for Java programs. *Formal Methods in Systems Design*, 24(2):129–155, Mar. 2004.
11. M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Proceedings of the European Conf. on Real-Time Systems - ECRTS'99*, pages 114–121, June 1999.
12. X. Liu, Z. Guo, X. Wang, F. Chen, X. L. J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI*, 2008.
13. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *ACM SOSP*, 2005.
14. B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.
15. A. K. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, June 1997.
16. V. Paxson, J. Kurose, C. Partridge, and E. W. Zegura. End-to-end routing behavior in the internet. In *IEEE/ACM Transactions on Networking*, pages 601–615, 1996.
17. P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.
18. S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. *IEEE Computer*, 1993.
19. T. Savor and R. E. Seviora. Toward automatic detection of software failures. In *IEEE Computer*, pages 68–74, Aug. 1998.
20. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.
21. K. Sen, G. Rosu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *Proceedings of TACAS 2004*, pages 123–138, May 2004.
22. K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *26th International Conference on Software Engineering (ICSE'04)*, pages 418–427, 2004.
23. O. Sokolsky, U. Sammapun, I. Lee, and J. Kim. Run-time checking of dynamic properties. In *Proceeding of the 5th International Workshop on Runtime Verification (RV'05)*, Edinburgh, Scotland, UK, July 2005.