Run-time Detection in Parallel and Distributed Systems: Application to Safety-Critical Systems

Beth Plale Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
{beth,schwan}@cc.gatech.edu

Abstract

There is growing interest in run-time detection as parallel and distributed systems grow larger and more complex. This work targets run-time analysis of complex, interactive scientific applications for purposes of attaining scalability improvements with respect to the amount and complexity of the data transmitted, transformed, and shared among different application components. Such improvements are derived from using database techniques to manipulate data streams. Namely, by imposing a relational model on the data streams, constraints on the stream may be expressed as database queries evaluated against the data events comprising the stream. The application in this paper is to a safety-critical system.

This paper also presents a tool, dQUOB, Dynamic QUery OBjects, which (1) offers the means for dynamic creation of queries and for their application to large data streams, (2) permits implementation and runtime use of multiple 'query optimization' techniques, and (3) supports dynamic reoptimization of queries based on streams' dynamic behavior.

1. Introduction

There is growing interest in run-time detection techniques as parallel and distributed systems grow larger and more complex. Run-time detection applied to such systems enables the analysis of the system's dynamic behavior for purposes of:

- evaluating system performance,
- recognizing current system states and/or predicting/forecasting future states, and
- validating system correctness in terms of its run-time operation.

System performance evaluation, as is done by performance measurement tools such as Paradyn [6], Falcon[4], or Pablo[11], employs run-time detection to gather information about program performance and resource usage, including the measurement of predetermined quantities like system utilization or program execution times, the analysis of application-specific quantities like the execution times of certain procedures or inner loops, or the collection of more complex information like the detection and evaluation of critical execution paths in parallel programs.Recognizing current system states to predict future states is demonstrated by network management tools [16] which attempt to understand network behavior in order to forecast future performance. Runtime validation techniques use data gathered at runtime to identify behavior deviating from some user-provided definition of correctness. Such deviant behavior may then be reported and/or corrected. Detecting hazard conditions in a safety critical system [12] is an example.

Runtime detection can be used as part of a larger adaptation strategy where collection and decision phases precede an enactment phase. For example, run-time detection might collect and analyze information to identify inappropriate behavior in a target system. An adaptation policy would then be invoked to adjust operation to improve performance or to meet its future goals. Adaptation strategies vary considerably in purpose. Algorithmic or parametric changes in real-time systems manage timing behavior in the presence of unpredictable programs or external environments; automated adjustments in distributed or high performance systems are carried out to improve the usage of scarce resources; and finally user-directed system changes, typically termed program steering, occurs.

1.1. Approach.

The runtime detection techniques we are developing are intended for distributed and parallel applications. The specific example of their use presented in this paper is the detection of hazards in continuous safety-critical applications. Our ongoing work is in applying these techniques to large-scale high performance applications running on multiprocessors and clusters of workstations, in the context of the Distributed Laboratories project described elsewhere[9]. Our approach to run-time detection imposes a relational database model upon the data arriving from the application. Specifically, we view the flow(s) of data from the application being observed to the analysis tool(s) as streams of events. Events may be extracted from files, received from sockets, or received as signals from an electro-mechanical sensor. When modeled in the context of our relational approach, an event is described as a database tuple [13]. All events of a given type are then members of the same relation, and relation membership is distinct (*i.e.*, no tuple belongs to more than one relation.)

By imposing the relational model on events, run-time detection reduces to the problem of query evaluation. A **query** is a language expression that describes data to be retrieved from a database. In our model, the event stream replaces the database as the data source. Specifically, events flow through a query evaluator in a pipelined fashion. If a particular set of events causes the query to be true, the query is said to be satisfied by that set of events. If applied to runtime validation, assuming a constraint is represented as a single query, a satisfied query equates to a constraint violation. If applied to performance enhancement, a satisfied query means the event or set of events met the criteria specified by the condition. A satisfied query can trigger such actions as performing computation, forwarding events, or generating a new event.

1.2. Discussion.

There are several advantages to a relational approach to describing and manipulating event streams. The rich and well-understood semantics of query languages permit the specification of complex conditions upon which events and event streams may be manipulated. We have investigated interesting runtime optimizations derived from known query optimization techniques. Finally, the resulting solutions to detection are scalable with respect to the size and complexity of the problems they address, and they offer means of adapting to changes in applications and their execution environments. Specifically:

- it is straightforward to specify detection conditions across multiple event streams and/or event types;
- conditions may be stated across both spatial and time properties of events;
- new event types may be created at runtime and queries may be changed dynamically, as well; and
- action routines may be executed when conditions are satisfied, thereby permitting the execution of application-specific computations as part of event processing.

Supporting scalability and adaptability is important for several reasons. First, run-time violations often involve multiple sources. For example, a constraint violation in an electro-mechanical system may involve a pump's pressure sensor, temperature gauge, and water valve sensor. The ability to handle conditions involving a large number of event types is one difference of our work compared to past approaches in violation detection that embed assertions in sensors or that use automatic code generation; both of which restrict conditions to attributes of a single task or single event type. Second, violations occurring in distributed systems often involve time. Therefore, scalability in terms of complexity of event streams is served by a query language that supports both relative time (*e.g.*, *b* happens after *a*) and logical or physical time (*e.g.*, *a* happens within 10 timesteps of *b*). Third, for large-scale distributed applications, any approach to detection should handle a large number of streams in a natural fashion.

Adaptability is important because realistic large-scale applications are typically dynamic, including the runtime creation and deletion of application components, the dynamic migration of components across underlying machines, and the use of alternative internal solution algorithms, data representations, etc. Our approach supports dynamic application behaviors in two ways. First, the database concept of a view is extended to event stream thus permitting the runtime creation of a new event type which can then serve as the input to a query. Second, the semantics of the query language presented in this paper permit the invocation of an action routine upon condition satisfaction. Action routines, having access to the events causing the violation, may perform useful application-specific analyses. New event types coupled with dynamically swapped queries and action routines provide interesting adaptation possibilities. this is part of our ongoing work.

To test the effectiveness of the relational approach to data stream analysis and manipulation, we have developed *Dynamic QUery OBjects* (dQUOB), a query evaluation tool for event streams. Its initial design [12] and its SQL-like, rule-based language [10] have been presented elsewhere. This paper focuses on the tool's dynamic capabilities and the performance results obtained

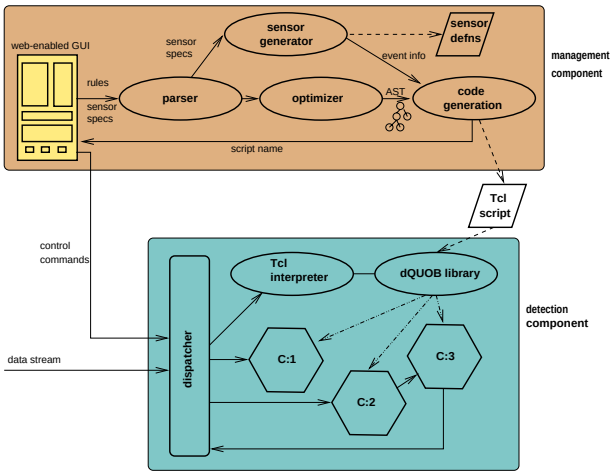

Figure 1. dQUOB components.

from applying dQUOB to a safety critical system.

The remainder of the paper is organized as follows. Section 2 provides a brief overview of the dQUOB tool. Section 3 discusses the safety-critical application used in our work and defines an example constraint. Dynamic constraint optimization is discussed in Section 4, followed by performance results in Section 5. Related work and the conclusion follows in Sections 6 and 7, respectively.

2. dQUOB Model

dQUOB is modeled as a set of nodes, queues between nodes, and a dispatcher controlling overall execution. A *node* is a collection of select, project, and join operations. Nodes are connected as a DAG where an arc exists between two nodes $\mathcal{A}$ and $\mathcal{B}$ if node $\mathcal{B}$ requires an input event type that is generated by $\mathcal{A}$.

A node implements one event-action rule. Rules are specified with the dQUOB rule language. The rule language draws heavily on the active database rule language Starburst [15] for its meta statements, and on the relational temporal query language ATSQL2 [14] for the event conditions. In this paper we give a single example of a complex constraint implemented as two rules. Further details on the rule language can be found in [10].

As shown in Figure 1, dQUOB consists of a management component and a detection component. The user interacts with the management component through a web-enabled graphical user interface, and through it, enters rules and sensor specifications, the latter to describe event sources. Sensor specifications are transformed by the sensor generator into sensor definitions which are subsequently linked with the application and accessed by user inserted instrumentation points. Rules are optimized before being transformed into calls to the dQUOB library. A rule becomes instantiated in the detection component through a control command issued to the dispatcher. Upon receipt, the dispatcher invokes the resident Tcl interpreter with the name of a script. The interpreter invokes the dQUOB library to instantiate operation and node objects that constitute the rule's executable form. Once instantiated, the rule registers itself with the dispatcher. Figure 1 shows three rules: **C:1**, **C:2**, and **C:3**. **C:1** and **C:2** take input from the data stream while **C:3** takes input generated by **C:2**. The arrow back to the dispatcher from **C:3** means the rule includes an action to be executed on behalf of **C:3** by the dispatcher.

The dispatcher controls dQUOB execution and as such, is responsible for linking rules, and for maintaining information about the current set of active rules and their relationship to one another. More importantly, however, the dispatcher services the data stream emanating from the distributed application components, from the management component, or from the detection component itself, as occurs during reoptimization, not shown.

2.1. Run-time Environment.

The communication substrate of dQUOB is shown in Figure 2. The application used for the paper is a threaded robotics simulation, where one thread simulates the operation of one robot. At instrumentation points in the application, threads invoke sensors to write event data to a shared buffer. The Falcon [4] monitoring substrate, running as one or more additional threads, extracts event data from the buffer, binary encodes it using the PBIO binary library, then forwards it via the DataExchange communication infrastructure [3].

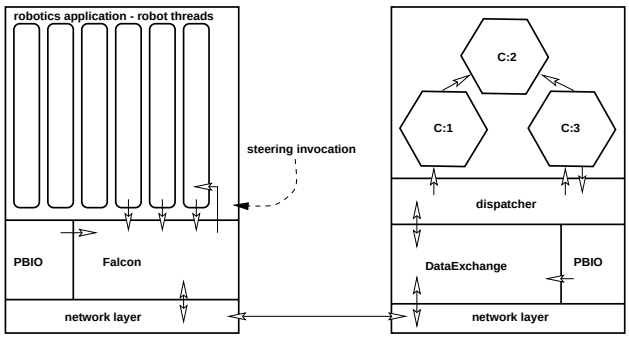

Figure 2. The communication infrastructure.

Monitoring events arriving at the analysis tool are de-

coded with PBIO before being passed to the event dispatcher. The dispatcher forwards events to the relevant rules. If a rule condition is satisfied, resulting in a steering action being triggered, as occurs for **C:3**, the rule will notify the dispatcher which will in turn pass the steering request to the application via DataExchange. Falcon receives and decodes this event before invoking the appropriate steering function embedded in the application code. This compile-time approach to instrumentation could be replaced by more dynamic instrumentation techniques such as binary editing [6].

3. Autonomous Robotics Example

The safety-critical application used in our work is a parallel multiagent reactive robotic system simulation.where a set of autonomous robots navigate toward a goal across unmapped terrain, each 'reacting' to stimuli in the environment. Some of the control schemes evaluated by this application have been tested with actual physical robots in field applications.

The control scheme on which we focus in this paper is called *forage*, where each robot in the robot group wanders the environment and upon encountering an attractor, moves toward the attractor, attaches itself, and returns the object to a specified home base. This is implemented as a FSM of three states: *wander*, *acquire*, and *deliver*.

Suppose a user wishes to specify that a violation occurs if a robot transitions to ACQUIRE state but does not subsequently transition to DELIVER within 10 timesteps. The constraint, appearing in Figure 3, is specified as two rules. **C:1** generates a derived event whenever a robot transitions from WANDER state to ACQUIRE state. The receipt of this event at **C:2** causes the nested query to be evaluated. Notice that **C:2** incorporates a real-time quantifier, `IN 10`, on the negated quantifier `NOT EXISTS`. The nested query fails if, from the time of arrival of the `AcquireEv` for 10 timesteps, no DELIVER event has been received. A failed nested query yields a satisfied outer query so a `NoDeliverEv` is generated. Additionally, a STEER action is executed to change the gain force value to $+0.2$ for the robot identified by AcquireEv.id. A STEER action results in a steering command being sent to the instrumented application.

```
CREATE RULE C:1 ON StateEv
IF
   SELECT AcquireEv s1.id s2.state
   FROM StateEv as s1, StateEv as s2
   WHERE
      s1.id = s2.id and s1.state = WANDER and
      s2.state = ACQUIRE and s1 meets s2
THEN


CREATE RULE C:2 ON AcquireEv, StateEv
IF
   SELECT NoDeliverEv
   FROM StateEv as s1, AcquireEv as a1
   WHERE NOT EXISTS IN 10 (
      SELECT
      FROM StateEv as s2, StateEv as s3
      WHERE
         s3.id = s4.id and s3.state = ACQUIRE and
         s4.state = DELIVER and s3 meets s4
   )
THEN
   STEER changeGoalGainForce AcquireEv.id +0.2
```

Figure 3. Constraint specified in two rules using the dQUOB event-action rule language.

4. Adaptive Query Evaluation

The adaptive capability of dQUOB makes it responsive to dynamic and long running applications. The adaptation reported in this paper replaces a rule with a more appropriate or efficient rule at run-time. A more appropriate rule might surface as the user gains deeper understanding of the system. More efficient rules might also occur under algorithmic control triggered by statistical data gathered about the event stream at run-time. This latter adaptation can yield a significant reduction in the number of events processed by a query. When statistical data about application data is considered during the ordering of a query's select operations, even a simple swapping of select operations can dramatically reduce the number of events processed by a node.

The former, user controlled adaptation, is provided through the management component shown in Figure 1 through which the user can add, remove, alter, activate, and deactivate rules. The algorithmic approach, called *dynamic constraint optimization*, is described in more detail in the next section.

4.1. Dynamic Constraint Optimization

Query plan generation applies well-known heuristics (*e.g.*, push selects below joins), then generates multiple plans based on indices available, order of joins, etc. A critical factor in determining the cost of one of these plans is the selectivity of the select conditions where *selectivity* is defined as the fraction of tuples satisfying the condition [8]. For databases, the selectivity of a condition can be computed during plan selection by scanning the table containing the relevant attribute. For the

event streams addressed by our work, selectivity estimates must be performed at runtime. Therefore, an important part of our research involves adapting selectivity estimation to an event-stream model.

Selectivity estimation is complicated by the fact that constraint violations often occur as spikes in the data, the implication being that we cannot assume a uniform distribution of data for event streams. This prevents us from using many of the simpler selectivity estimates that assume such a distribution [8].

Implementing selectivity estimates involves a two-fold approach:

- using random sampling at run-time to compute selectivity estimates, and
- building support for reoptimizing a query at runtime by swapping one query for its more optimal cousin.

4.2. Selectivity Estimation

Selectivity estimates (*e.g.*, SEL(sex="female") = 0.5 and SEL(salary $>$ \$1,000,000) = 0.02) obtained by earlier simple statistics such as minimum and maximum values are accurate only if attribute values are uniformly distributed. Those obtained by commonly-used equi-width histograms, where the number of elements in a bucket varies across buckets, suffer the same limitation [7].

Equi-depth histograms, on the other hand, have been shown to provide more accurate estimates for non-uniformly distributed data [8]. Equi-depth histograms, which fix the height of each bucket instead of its width, are built by sorting the records in a table, and partitioning the records amongst the buckets. Since sorting a table is not possible, we adopt a sampling technique to approximate the distribution of values [8] and use a non-parametric statistic, Kolmogorov's statistic,for the analysis of estimation errors. Kolmogorov's statistic states the relationship between confidence and sample size. For instance, suppose the sample size is 1064 tuples and β is the fraction of tuples in the sample with *attr* $<$ *v*, then with confidence 99% the fraction of the tuples in the entire relation with *attr* $<$ *v* is in the interval [β - 0.05, β + 0.05]. The sample size does not depend on the number of tuples in the relation.

4.3. Query Reoptimization

Dynamic constraint optimization involves cooperation among several components. As shown in Figure 4, the rule **C:1** collects statistical data about its attributes as equi-depth histograms. The rule triggers reoptimization initially after a histogram has been established then

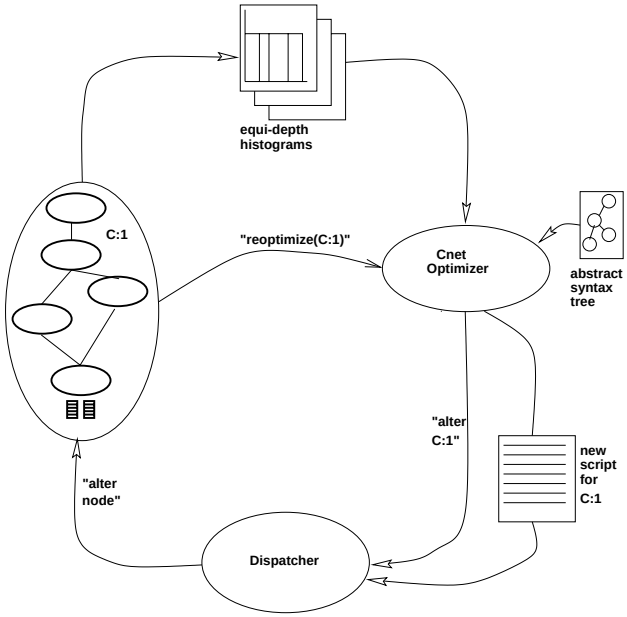

Figure 4. Dynamic rule optimization.

thereafter on a periodic basis. The optimizer, running as a separate thread, retrieves the appropriate abstract syntax tree based on the name of the invoking rule. Reoptimization can now occur with selectivity data present which will likely cause a more optimized version of the rule to be generated. The optimizer generates a new script and issues a command to the dispatcher to alter the existing rule. The dispatcher replaces the rule with the more optimized version, while preserving its accumulated state to prevent missed violations.

5. Evaluation

5.1. Microbenchmark

We first undertook the timing of the individual operations (*e.g.*, select, project, join) and used these times to compute a total execution time for a query. These microbenchmark values can be used to predict the execution time of a query and can also be useful to highlight analysis overhead.

The benchmark numbers are obtained using a set of twelve constraints, with the set composition varying depending on the operation being measured. To measure the *gate* operation, every constraint consists of a single gate operation accepting a single primitive event. Though the gate operation serves as both an entry-point to the node and point-of-control for the node, during testing the operation simply dequeues an event and re-

turns. In measuring the remaining operations, each constraint consisted of a gate operation and at least one operation of the type being measured. For example, of the twelve constraints in the *select* benchmark, eight have one select operation and four have two; eight accept a primitive event while four accept a derived event.

operation	*execution time (microseconds)*
gate	4.7
projection	8.6
selection	1.6
join	0.5

Table 1. Microbenchmark results.

The microbenchmark results are shown in Table 1. Results were obtained using a set of 16872 events generated by the robotics application and run on a Sun UltraSPARC 1. The time shown for the gate operation may be misleading. The gate operation in fact does very little; the time shown includes the overhead of the dispatcher. As is evident, projection is a costly operation with respect to the others, but the cost is reasonable considering that projection creates a new event for every event it receives.

To verify the reasonableness of the benchmark values, we compare the computed time versus measured time for a *typical mix* of realistic queries. Although the individual queries in the mix are simple, each is a meaningful query, the set accepts multiple event types, both from sensors and from other queries. The structure of the typical mix is shown in Figure 5. The queries C:1, C:3, C:6, C:8, C:10, and C:7 accept input from sensors while the remaining queries accept derived events.

As shown in Table 2 the typical mix resulted in 362,569 operations (*i.e.*, number of times a single select, project, etc. operation was executed). The measured execution time of 9% greater than the computed execution time indicates there is activity in the analysis tool for which the benchmark values are unable to account. We suspect this is due, in part, to caching effects.

total ops	*measured execution time (seconds)*	*computed execution time (seconds)*
362,569	1.51988	1.39385

Table 2. *Typical* **mix.**

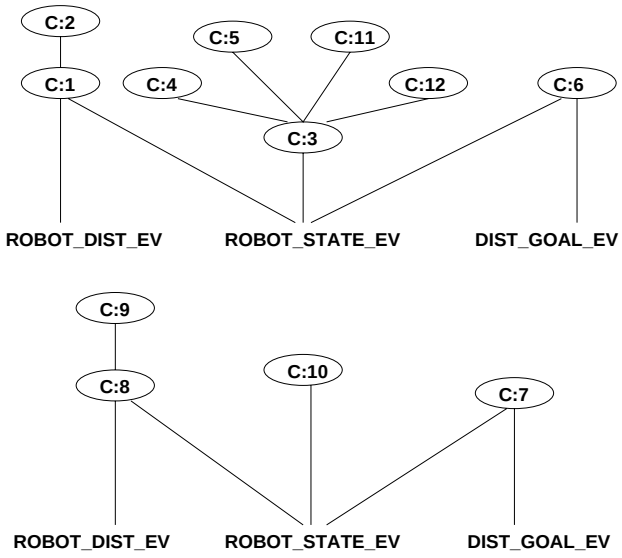

Figure 5. dQUOB for *typical* **mix.**

5.2. Performance Gains of Optimization

The evaluation criteria used in measuring the benefit of optimization is total time to execute a query, measured from the time the first event is available until the last event is received. The queries used execute no actions. The results include the total execution time and number of operations performed for the constraint in optimized and unoptimized forms. Each constraint is applied to the event stream of 16872 events.

Optimization A	*optimized*	*unoptimized*
execution time (secs)	0.14427	1.08329
number ops	34,070	865,411
% by op (gate/project/select/join)	.49/0/.51/0	.02/0/.11/.87

Table 3. Optimization A, a complex constraint.

Query **A**, shown in Table 3 accepts three input event types. The optimization which emphasized pushing selects down the parse tree, yielded a 87% reduction in time and a 96% reduction in the number of operations. As can be seen from the breakdown by type, in the unoptimized version 87% of the operations were attributed to join; the number of joins was reduced to zero in the optimized version. This indicates the optimization policy of pushing selects is effective in reducing the number of events reaching joins.

Filter A	*filter*	*no filter*
execution time (secs)	0.22191	0.23224
number ops	33,345	72,171
% by op (gate/project/select/join)	.33/.33/.33/0	.31/.07/.62/0

Table 4. Effects of factorization.

5.3. Inter-Rule Optimization

Greater gains in optimization can be achieved when inter-rule optimizations are considered as well. *Factorization* is the factoring out of a subexpression common to two or more queries; Table 4 illustrates the effects. The results seem to indicate that little benefit is accrued as a result of factorization. Though the total number of operations is reduced by more than half in the filtered version, the execution time drops only 4.6%. Insight to the problem can be gained by looking at the operations breakdown, particularly at the percentage attributed to projection. In the no-filter case, projects make up 29% of the total time and 7% of the total operations. In the filter case, projects account for a significant 67% of the total time and 33% of the total operations.

The time gained in the rather substantial filtering that took place (not evident by the numbers presented) was partially lost through additional projection operations which we know from microbenchmarking are costly. The benefit of factorization, we conclude, depends on a number of variables: first, the filtering capacity of the factored-out select condition weighed against the cost of introducing at least one project. Since filtering capacity is determined by selectivity estimates, factorization decisions should be deferred to run-time. The second factor is the location of the filter. Were the filter located closer to the source, the cost of filtering could be weighed against the benefit of lower latency and lessened network demand.

6. Related Work

Afjeh *et. al* [1] propose a monitoring and control system for observing and steering a parallel propulsion system simulation. Analysis employs an expert system where satisfied rules trigger steering commands. An expert system may not be appropriate for our work for two reasons. First, distributed analysis is not easily accomplished with expert systems. Second, the rules which encapsulate domain-specific knowledge can require significant setup times and much up-front knowledge about potential violations.

Brockmeyer and Jahanian [2] incorporate a monitoring and assertion checking tool into the Modechart Toolset (MT). Assertion checking is performed on trace data from symbolic executions of real-time specifications . Although similar to ours in its support of temporal and complex specifications, this work cannot be directly applied to run-time detection.

Snodgrass [13] develops an information based approach to modeling program behavior that treats monitoring information (runtime data, states of processes, states of processors, messages, etc.) as relations. Prior applications of this formalism were geared toward performance evaluation where the focus is collection and organization of information for what is often postmortem analysis. Run-time detection, on the other hand, is oriented toward discarding information as early and often as possible. This focus shift makes the technique more tractable than earlier work. In addition, prior approaches were static, that is, they required that all constraints be known at compile time. Given the exploratory, 'what-if' potential of safety constraints, application of the relational model to hazard detection must be accompanied by adaptation techniques.

Leveson's work in the early 1980's [5] is early recognition of the need for run-time checking for hazard prevention. The synchronous approach doesn't scale well to detection for distributed applications.

7. Conclusion and Future Work

This paper presents results of our research on run-time detection achieved through imposing a relational model on the event stream. The detection techniques we are developing are broadly applicable to distributed and parallel applications, but the specific domain to which we apply the work in the context of this paper is safety critical applications. Existing approaches to safety, often based on formal techniques, cannot always provide the required assurance of safety. Detection approaches can complement formal techniques to provide greater levels of assurance.

Our ongoing work is applying the formalism to large-scale high performance computations running on multiprocessors and clusters of workstations. As described in the paper, query evaluation can be beneficially applied to data stream management in a Distributed Laboratory environment.

The relational approach offers a rich and well-understood query language, and known optimization techniques that make detection scalable with respect to the size and complexity of the problems they address. Further, our solution offers a means of adapting to changes in applications and their execution environments by the ability to create new event types and

add/replace rules at run-time.

More specific ongoing work includes distributing the analysis to enable communication between constraints irrespective of constraint location and evaluating the selectivity optimization algorithm. Additionally, research continues into the fruitfulness of selectivity estimates. These statistics offer an interesting potential for making statements about the selectivity of an entire query and are particularly useful applied to filtering in the Distributed Laboratories environment. Finally, the temporal aspects of the language are being more richly exploited through application to additional applications.

References

[1] A. Afjeh, P. Homer, H. Lewandowski, J. Reed, and R. Schlichting. Development of an intelligent monitoring and control system for a heterogeneous numerical propulsion system simulation. In *Proc. 28th Annual Simulation Symposium*, Phoenix, AZ, April 1995.

[2] Monica Brockmeyer, Farnam Jahanian, Connie Heitmeyer, and Bruce Labaw. An approach to monitoring and assertion-checking of real time specifications in Modechart. In *Workshop on Parallel and Distributed Real-Time Systems*, April 1996.

[3] Greg Eisenhauer, Beth (Plale) Schroeder, and Karsten Schwan. DataExchange: High performance communication in Distributed Laboratories. In *Proceedings Ninth IASTED Int'l Conference on Parallel and Distributed Computing Systems*, October 1997.

[4] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, Aug. 1998.

[5] Nancy G. Leveson and Timothy J. Shimeall. Safety assertions for process-control systems. In *Proceedings 13th Int'l Symposium on Fault Tolerant Computing*, pages 236–240, June 1983.

[6] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. Paradyn parallel performance measurement tools. *IEEE Computer*, 28, November 1995.

[7] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings ACM SIGMOD Conference*, pages 28–36, June 1988.

[8] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings ACM SIGMOD Conference*, pages 256–276, June 1984.

[9] Beth Plale, Greg Eisenhauer, Karsten Schwan, Jeremy Heiner, Vernard Martin, and Jeffrey Vetter. From interactive applications to Distributed Laboratories. *IEEE Concurrency*, 6(2), 1998.

[10] Beth Plale and Karsten Schwan. Language issues in hazard detection using queries. Technical Report GIT-CC-97-36, College of Computing, Georgia Institute of Technology, 1997. http://www.cc.gatech.edu/tech_reports.

[11] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. *An Overview of the Pablo Performance Analysis Environment*. Department of Computer Science, University of Illinois, 1304 West Springfield Avenue, Urbana, Illinois 61801, November 1992.

[12] Beth (Plale) Schroeder, Sudhir Aggarwal, and Karsten Schwan. Software approach to hazard detection using on-line analysis of safety constraints. In *Proceedings 16th Symposium on Reliable and Distributed Systems*, pages 80–87. IEEE Computer Society, October 1997.

[13] Richard Snodgrass. A relational approach to monitoring complex systems. *IEEE Transactions on Computers*, 6(2):156–196, May 1988.

[14] Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen, and Andreas Steiner. Adding valid time to SQL/temporal. In *ISO/IEC JTC1/SC21/WG3 DBL MAD-146r2*, November 1996.

[15] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems*. Morgan Kaufmann, 1996.

[16] Rich Wolski. Dynamically forecasting network performance to support dynamic scheduling using the network weather service. In *Proceedings 6th High-Performance Distributed Computing (HPDC6)*, August 1997.