Abstract

Safety-critical systems are pervasive in our daily lives. Accompanying the growth of these systems is an increasing awareness of the potential dangers involved. As a result, there has been recent realization among researchers that it is important to consider entire systems and their safety characteristics, going beyond individual embedded or real-time system components. This new focus presents an opportunity for a new approach to software safety, one that can accommodate heterogeneous distributed systems that may contain COTS components and may consist of components not all of which were designed to be used in safety-critical settings.

In response to this need, we have developed a software hazard detection tool that we argue increases the safety level of continuous safety critical systems. In this paper we discuss the tool’s language and code generation. The appendix contains the language syntax and a full sample of the generated code.

Keywords: software safety, safety-critical systems, on-line monitoring, continuous distributed systems, query optimization

1 Introduction

1.1 Safety-critical Systems

Safety-critical systems are pervasive in modern society. A safety-critical system is any system wherein a problem or hazard occurring in the system could potentially compromise economic, property, or personal safety. Traditionally, safety methods have been directed toward well-defined system components, such as embedded or real-time components. Definitions of safety concerned system output or timing correctness, with an emphasis on correct and predictable performance at the task and communication levels. Examples of such embedded components include: digital flight control systems, nuclear power plants, collision avoidance systems on board aircraft, and computerized signaling systems for controlling commuter trains. There are other systems that are clearly safety-critical but that deviate from this traditional set of safety-critical systems, such as cardiac care products like patient monitoring devices for monitoring an intensive-care patient’s vital signs or implanted devices like defibrillators. In both cases again, safety may be defined as the correct and timely operation of embedded system components. Alternatively, correctness rather than timeliness is important for the ammunition control software (ACS) system which manages ammunition holdings. ACS is clearly safety-critical in that incorrect storage combinations can lead to massive explosions.

There is recent realization in the broad community of researchers concerned with safety-critical systems that it is important to consider entire systems and their safety characteristics, thereby going beyond considering individual embedded or real-time system components. This is due to the fact that computers in safety-critical roles are pervading all areas of our lives and that our understanding of the potential dangers of this pervasiveness is increasing. The implication of this growth is that an increasing number of software components that were never intended to be safety-critical are used in safety critical settings[2], as illustrated by an emergency room physician accessing a remotely located patient record for information essential to an
accurate diagnosis. A delay in responding to the request could potentially endanger the patient. Similar examples exist for transportation systems, financial systems, and electronic, telephone and telecommunications networks. Consider the Automated Vehicle/Highway System (AVHS) [16], a transportation system in which car-like vehicles travel in strictly controlled platoons so as to minimize the distance between each vehicle. One obvious safety-critical issue is crash prevention, particularly when vehicles change lanes or merge on or off the road. Similarly, computerized financial systems are considered safety critical not because of the threat to personal safety, but because of the potentially catastrophic harm to the economy should such a system fail. Such a definition of safety in terms of economic damage strengthens the importance of safety considerations for electronic, telephone, and telecommunications networks since loss of service removes the capability to summon emergency services. Similarly, loss of long-distance service can cause serious disruption of business activities and an ensuing loss of revenue [14].

We distinguish this broader class of safety-critical applications, which we call continuous safety-critical systems, from the more traditional safety-critical systems by the following characteristics:

- they are long-running,
- they consist of components distributed over multiple physical resources,
- they may evolve, and
- they may include commercial off-the-shelf (COTS) components or components not designed to be operated in safety-critical settings.

A long-running system may be up for weeks, months, or even years so it is highly likely that the nature of potential hazards would change over time as the environment changes. The components of this class of system can be geographically distributed, such as the emergency room physician accessing a remote patient record database. These systems may evolve in the sense that new tasks/objects may be added or removed during execution. Finally, these systems may include COTS components, such as the remote database management system of the example above. Such a component may not have originally been intended as a safety-critical component so techniques to achieving an intrinsically safe system, often employing formal methods, were likely to not have been employed or were difficult to employ due to system distribution or complexity.

Moreover, unlike real-time systems, continuous safety-critical systems are time dependent but only some of their components may have real-time constraints. The remainder of the system does not have the strict guarantees required of a real-time system, and its effort is considered to be best-effort as opposed to guaranteed-response [11]. Finally, unlike implanted devices, which are inaccessible once installed (e.g., defibrillators), continuous safety-critical systems are accessible from the 'outside' in terms of an ability to expand selected component functionality or even affect component operation during execution.

### 1.2 Hazard Detection

Our research addresses continuous safety-critical systems by 1) assuming that safety violations may occur and 2) aiming to cope with such violations by detection them. In other words, we aim to achieve safer software through a software hazard detection approach. Hazard detection is performed using the Cnet tool constructed as part of our research, which offers a language-based approach to on-line hazard detection that allows the user to specify constraints in the form of queries in a temporal database query language.

The architecture of the system is shown in Figure 1. It consists primarily of a user interface, sensor and constraint code generation component, and run-time environment. Control flow begins at the user interface where the user enters constraints and/or sensor specifications. The interface invokes the parser which parses the descriptions then takes different actions depending on whether the entry is a constraint or a sensor description. Sensor descriptions are passed to the sensor generator which generates sensor code which it writes to a file that is subsequently compiled with the application. The generator also passes pertinent sensor information to the code generation component of query processing. Conversely, the parser converts constraints into an abstract syntax tree (AST) which it then passes to the optimizer. The optimizer reorders the AST in accordance to optimization rules before invoking code generation. Code generation uses the AST and sensor information to generate a Tcl script. It then signals its completion by passing the name of the script file back to the user interface.

The user also selects a <hostname, port> pair from a list of available machines. The user interface responds to compilation completion by generating a remote call to the machine to create an analysis component consisting of a dispatcher, Tcl interpreter, and the Cnet library. Passed to the analysis component
is the name of the Tcl script returned by code generation. The dispatcher, once created, invokes the Tcl interpreter with the name of the file. The interpreter processes the script, invoking routines in the Cnet library which allocate space and create executable representations for a query. Once a query is created, it registers its existence with the dispatcher.

Cnet addresses the unique needs of hazard detection for continuous safety-critical systems in a number of ways. First, the interface between the compiler and library in the form of a script makes the approach well suited to long running and evolving applications by facilitating constraint set modifications. Adapts invoked via the user interface alter the constraint set (e.g., add, remove); algorithmic adaptations based on statistical run-time data also alter the constraint set by triggering query reoptimization. This dynamic behavior is key to the analysis tool’s responsiveness to the long running and evolving nature of continuous safety-critical systems. Second, the language-based approach allows for the specification of a general set of hazards, including temporal constraints and constraints with complex conditions specified over multiple components. Additionally, the language-based approach enables constraint optimizations to enhance performance, either dynamically or during constraint creation. Finally, the external nature of the approach facilitates efficient use with distributed application components as the analysis can be done on a machine independent from the application components. The external nature also allows the tool’s use regardless of the language with which the safety-critical application was written, subject to the needs of instrumentation. The analysis tool is able to analyze any uniform event stream, regardless of source.

The remainder of the paper is organized as follows. In the following section we describe a sample continuous safety-critical application to a level of detail sufficient to understand the example constraints. In Section 3 we describe the rule language and in Section 4, constraint generation. The run-time architecture and implementation is discussed in Section 5. The paper concludes in Section 6 with a discussion of future work.

2 An autonomous robotics application

The sample application used in our work is a multiagent reactive robotic system simulation [1] where a robot performs one of three tasks: forage, consume, and graze. During forage, a robot wanders around
looking for attractors. Upon encountering an attractor, it moves toward the attractor, attaches itself, and returns the object to a specified home base. The mass of the attractor dictates how quickly the robot can carry it; a heavier attractor means slower speed. Robots can cooperate to move attractors. During consume, a robot also wanders around, looking for attractors. Unlike forage, though, after attachment a robot performs work on the object in place. The time required to do in-place work is proportional to the mass of object and the number of robots cooperating to consume the attractor. In the final task, graze, discrete attractors are not involved; the object is to completely cover, or visit, the environment (akin to mowing the lawn). A robot searches for an area not grazed, moves toward it, then grazes until the entire environment (or some percentage thereof) has been covered.

A robot is implemented as a three state finite state machine with the state selection dependent on the current task. For example, in the forage task a robot can be in wander, acquire, or deliver state. In wander state, the robot roams freely. It transitions to acquire state when it has detected an attractor and to deliver state when it is returning the object to the home base. Once the object has been delivered, transition is made back to wander.

A robot reacts to its environment using schema-based control. That is, the robot’s movements are determined by the repelling and attracting forces of the robots, the obstacles, and the goals within the robot’s perception. These independent forces are summed and normalized before being actuated. One of the schema-based parameters is the robot repulsion gain, the amount of repelling force a robot applies toward another robot to maintain a separation. When wandering, the repelling force is relatively high so the robots will spread out to cover a wide area. During acquire, on the other hand, the repelling force is lessened so the robots can converge to jointly ‘carry’ a canister.

Other parameters include the number and location of robots, obstacles, and goals. Parameters are stored in shared maps of the world. Each robot can see only the portion of each map describing objects in its immediate proximity. This limited view is likened to a human having knowledge of only his/her immediate surroundings. In the current threaded version of the simulation, with each robot executing as a separate thread, communication is through the global maps residing in shared memory.

2.1 Hazard detection in the robot application

Meaningful constraints and actions can be specified in the context of an example. For instance, suppose the group of robots are tasked with digging up drums filled with radioactive material that are buried over a wide area and moving them to a central site on higher ground. In retrieving a drum, one of the robots is splashed by the thick, murky contents and begins emitting radiation. The user might desire to keep the other robots away from the contaminated robot. So in devising a constraint, the user specifies a hazard based on the concept of a danger zone, a region around the contaminated robot in which a robot approaching to help is in a danger but not imminently so. The user then wants to be notified when an approaching robot is within 10 ft. of the contaminated robot even though the danger of picking up contaminated debris from the robot may not occur until the approaching robot is within, say, 5 ft. With this constraint, a sufficiently large work area, and a large enough number of attractors to keep the robots working largely independently, the user may be confident enough to allow work to continue even after a single robot becomes contaminated. The constraint for the scenario just described is as follows:

• (C:1) The radioactivity level of a robot exceeds 200 roentgens per hour (R) and another robot approaches within 10 ft. of the radiating robot.

Several other meaningful constraints can be specified on robot behavior:

• (C:2) If a robot transitions from wander state to deliver state (without transitioning into acquire state first), then disable that robot.
• (C:3) If a robot transitions to acquire state but does not transition to deliver within 10 timesteps, then increase the move to goal gain for that robot.
• (C:4) If the radiation level of a robot in acquire state exceeds 200R, then increase the radiated robot’s other robot repelling force and activate constraint C:1.

Constraints can also have a management focus such as the following which gathers statistical information for evaluating performance:

• (C:5) If during forage task, the average amount of time spent by robots in the wander state exceeds a threshold while progress toward a goal is less than some minimum, then trigger a violation notification.
That is, the user may be interested in knowing when the average amount of time a robot spends wandering as a function of the total amount of time working exceeds some reasonable estimate. As long as the value is reasonable, the user is willing to let the event pass without being notified. The examples given above represent the types of constraints that can be supported. In the following section we introduce a language to specify these constraints.

3 Language description

A temporal relational query language, given its declarative style, optimization potential, temporal capacity, and ease of use is a highly suitable choice as a hazard detection specification language. The language we use is a relational query language incorporating features of the active database rule language Starburst [21] for its meta statements, and a subset of the relational temporal query language ATSQL2 [20] for specifying constraint conditions. We have extended ATSQL2 in a minor way as discussed in [3] to allow for the specification of real-time properties.

The user specifies desired active behavior with rules. These rules are triggered by the occurrence of an event. For example, a rule in an active database system may state that whenever an employee's bonus is increased by more than 100, that employee's rank is increased by 1. The rule would be triggered by the event of an update to the employee's bonus. Starburst [21] provides a rich set of semantics for specifying and managing rules, for prioritizing rules, and for inducing a partial ordering on the set of defined rules. Our language adopts the create rule, alter rule, drop rule, activate rule, and deactivate rule.

The create rule statement, used to create a new constraint, has the following syntax:

CREATE RULE name ON event-type-list
IF constraint-condition
THEN action-list

where name names the rule and event-type-list lists the event types (or relations) required to process the constraint. Constraint-condition specifies the condition to be checked when the rule is triggered by an incoming event. The condition is specified with the temporal query language ATSQL2. ATSQL2 is a superset of SQL, a variant of TSQL2 [19] and is currently being proposed for incorporation into SQL3 [20]. The syntax of the constraint condition is given in Section 3.1. Action-list specifies a list of actions to be executed when the constraint condition evaluates to true. Its syntax is given in Section 3.2.

The second rule, the alter rule, is used to redefine an existing rule:

ALTER RULE name ON event-type-list
IF constraint-condition
THEN action-list

where name and event-type-list identify the constraint to be replaced, and constraint-condition and action-list identify the new condition and actions. The alter rule is rejected if event-type-list does not match the existing event type list. The third rule, the drop rule, is used to delete an existing rule. Its syntax is:

DROP RULE name ON event-type-list

where here again, input-event-list is used to authenticate the rule. The rule is rejected if the input event list and existing event type list do not match.

The final two rules change the state of a rule:

where name and event-type-list name a rule. The deactivate rule temporarily deactivates a rule. A deactivated rule can subsequently be reactivated by an activate command. A rule when created is active and remains active until explicitly deactivated or removed.

The create rule is illustrated in Figure 2. The create rule statement creates a rule named C:1 with event sources as RobotRadEv and RobotDistEv. Event sources can be a sensor or a previously defined rule. The
DEACTIVATE RULE name ON event-type-list

ACTIVATE RULE name ON event-type-list

CREATE RULE C:1 ON RobotRadEv, RobotDistEv
IF
  SELECT RadiatedRobotEv d.idSelf r.rad
  FROM RobotRadEv as r, RobotDistEv as d
  WHERE
    r.id = d.idSelf and d.dist < 10 and
    r.rad >= 200
THEN

Figure 2: Radiation level of robot exceeds 200 R and another robot approaches within 10 ft. of radiating robot.

IF statement delineates the rule’s condition, and is described next. The THEN statement, which delineates the action list contains no actions.

3.1 Condition specification through ATSQL2

ATSQL is a high-level declarative language containing statements for data definition, query, and update; for defining views on the database and for embedding statements into a general-purpose programming language such as C or C++. ATSQL2 is a prototype language developed to test the proposed addition of temporal support to SQL. An important criteria in language design was upward compatibility. That is, all application code had to work with the new system with exactly the same functionality as the existing system. To accommodate upward compatibility and smooth the transition from a non-temporal environment to a temporal one, the designers proposed four levels of temporal functionality. Levels are distinguished from one another largely through the addition of syntactic variables and statements. Specifically, the level of temporal support desired by a programmer is specified by the presence or absence of qualifying words in the manipulation statements. We omit the qualifying words from our language because the syntactic overhead required to support several levels is not required. In fact, we support temporal operators, which is at the highest of ATSQL’s levels.

The constraint condition is specified with the ATSQL2 SELECT construct:

SELECT relation attribute-list
FROM table-list
WHERE condition

where
- relation names the new relation,
- attribute-list is a list of attribute names whose values are to be retrieved by the query,
- table-list is a list of the relation names required to process the query, and
- condition is a conditional (Boolean) search expression that identifies the tuples to be retrieved by the query.

The constraint in Figure 2 selects the events that satisfy the condition of the WHERE clause, then projects the result on the d.idSelf and r.rad attributes for the derived event RadiatedRobotEv listed in the SELECT clause.

Whereas the SELECT clause specifies the projection attributes, the WHERE clause specifies the selection condition. The condition r.id = d.idSelf is a join condition on the relations RobotRadEv and RobotDistEv. Since the same name can be used for two or more attributes as long as the attributes are in different relations, the language supports aliases, a feature to declare alternative names for a relation. The syntax for an alias is the alias name followed by a dot (e.g., r.id) preffacing an attribute. In the example, r is an alias for RobotRadEv and d for RobotDistEv. The aliases can appear in the WHERE and SELECT clauses and in the action statement.
Several binary temporal operators are provided: `meets`, `precedes`, `overlap`, and `contains`. The `meets` operator is shown in Figure 3. A violation for $C:2$ occurs if a robot transitions from WANDER state directly to DELIVER state (without having transitioned to ACQUIRE). Specifically, the violation occurs if an $s_1$ event is received for a robot in WANDER, an $s_2$ event is received for the same robot in DELIVER and the logical time of event $s_1$ is one timestep less than the logical time of $s_2$. That is $s_1\.logicaltime + 1 = s_2\.logicaltime$

CREATE RULE $C:2$ ON StateEv
IF
SELECT BadTransitionEv s1.id
FROM StateEv as s1, StateEv as s2
WHERE
  s1.id = s2.id and s1\.state = WANDER and
  s2\.state = DELIVER and s1 meets s2
THEN
STEER DEACTIVATE StateEv\.ID

Figure 3: Robot transitions from WANDER to DELIVER.

CREATE RULE $C:3$ 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 $C:3$.1 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 4: Robot transitions to ACQUIRE but does not transition to DELIVER within 10 timesteps.

Nested queries are queries nested within another query. Referring to Figure 4, suppose a user wishes to specify that a violation occurs if a robot transitions to ACQUIRE state but does not transition to DELIVER within 10 timesteps. The constraint is specified as two queries. $C:3$ generates a derived event when a robot transitions from WANDER state to ACQUIRE state. The receipt of this event at $C:3\.1$ causes the nested query to be evaluated. Notice that $C:3\.1$ incorporates a real-time component as given by the 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.
due to the negated existential quantifier, so a NoDeliverEv is generated. Additionally, a STEER action is executed which invokes the changeGoalGainForce function in the robot application for the robot named by the attribute AcquireEv.id and with the goal gain force value of +0.2. Action statements are described in the following section.

3.2 Action statements

Action statements are commands to the run-time environment. The analysis tool provides a basic set of allowable actions such that when taken alone or combined, the actions allow the user to effect a desired behavior. Four actions are supported:

EXEC_FUNC function-name arg1 arg2 ...
STEER function-name arg1 arg2 ...
ACTIVATE constraint-name
DEACTIVATE constraint-name

where function-name names a function resident in the application, in the case of STEER, or the analysis tool, in the case of EXEC_FUNC and constraint-name names a constraint (e.g., C:1). We make the assumption that the user defined code in the analysis executable is small and simple. Without this assumption, we could not make any guarantees about meeting the low-latency demands of a safety-critical application. The arguments arg1, arg2, ... contain parameter values passed to the function.

EXEC_FUNC invokes a user-defined function in the analysis tool that might do something simple, like causing a bell to ring or a message to be printed to an operator console or do something more complex, like collecting statistics or performing computations on the data. STEER results in the execution of a user-defined function in the application. ACTIVATE and DEACTIVATE cause the activation or deactivation of the node. The former causes a disabled constraint node to become active whereas the latter deactivates an active constraint node. A deactivated node neither processes events nor evaluates its query. A node when created is active and remains active until explicitly deactivated or removed. A deactivated rule can subsequently be reactivated.

CREATE RULE C:4 ON RadLevelEv
IF
    SELECT TooHighEv e1.id e1.rad
    FROM RadLevelEv as e1
    WHERE
        e1.rad > 200 and e1.state = ACQUIRE
THEN
    STEER changeOtherRobotAcquireForce RadLevelEv.id +0.01
    ACTIVATE C:1
    DEACTIVATE C:4

Figure 5: Radiation level of robot in ACQUIRE exceeds 200R.

The constraint shown in Figure 5 illustrates the use actions. A constraint violation occurs if the radiation level of a robot in ACQUIRE state exceeds 200 roentgens per hour. The first action causes invocation of the changeOtherRobotAcquireForce function in the application for the robot given by the first parameter. The integer value increases the robot’s acquire force, making it less likely that other robots will move toward the radiating robot. The second action causes the constraint C:1 to be enabled; the third causes the disabling of this constraint, presumably because C:1 replaces the need C:4.

Any approach, such as the one discussed in this paper, that allows dynamic constraint addition must address conflicting constraints. That is, eventually an added constraint will conflict with an existing one. The effect of conflicting constraints is that one or the other will continuously be violated; perhaps not a
desirable behavior from the point of view of the user. We provide the ACTIVATE and DEACTIVATE statements largely as a means for managing constraint conflict.

The activate and deactivate actions may also be useful for loosely hierarchical error recovery [4]. For example, when a robot encounters an obstacle in its path, its first response could be to wait some amount of time in hopes that the obstacle will move. If this simple error recovery fails, its second response would be to determine a new route. This recovery could be achieved by specifying two constraints; the first with an action of robot waiting. If the violation continues to occur after a specified period of time, the second constraint is activated. The second constraint has as its actions to calculate a new route and to deactivate the earlier constraint.

3.3 Adapting query evaluation to an on-line environment

The significant issue in on-line query evaluation is that there is no physical database against which a query is evaluated; instead the database is purely conceptual. Still, the conceptual type influences design decisions so we consider two types: temporal databases and valid-time databases.

Temporal databases are databases that support both valid time and transaction time. Valid time is the time known and maintained by the application. It can be a physical time, logical time, or both. Transaction time, on the other hand, is the time at which a tuple or set of tuples entered the database [17]. Transaction time provides a history of all past states of the database and is useful as a record of corrections or modifications to past database states. Valid-time databases [10], on the other hand, support valid time but not transaction time. It has been determined in [18] that for purposes of monitoring, a valid time database is sufficient. The reason for this is apparent when one considers that in an on-line environment, events flow through a collection of constraints and, if not retained by one or more constraints, will be discarded. In such a setting where it is not possible to update or delete a tuple (i.e., event) corrections to past database state cannot be made. This inability to update past database state removes the need for a transaction time database. Additionally, in an environment where a physical database exists, a query is executed periodically or upon user demand, and a set of tuples satisfying the query is returned. In contrast, in a on-line environment, a query is in essence executed every time an event arrives. What keeps this characteristic from resulting in a wholly inefficient execution is that a constraint will by nature reject the majority of the events it receives. And with good optimization heuristics, the events will be rejected as early in the evaluation process as possible.

We restrict the language to a subset of ATSQL2 largely for efficiency reasons. For example, a missing WHERE clause indicates the absence of a condition on tuple selection, hence all tuples of the relation specified in the FROM clause qualify and are selected for the query result. We require the presence of a condition to limit the number of tuples flowing through the analysis tool. Additionally, to retrieve all the attribute values of the selected tuples, SQL allows the use of an asterisk (*) which stands for all the attributes placed on the SELECT line following the derived event name. We require the attribute set to be explicitly named.

4 Constraint Generation

Just how a query can be evaluated against incoming event streams instead of against tables in a database becomes more clear when one looks at the query’s relational algebraic form. The relational algebraic form is useful because it is prescriptive in nature and implicit in the expression is a procedure for solving the problem. There is no loss of generality in expressing an ATSQL2 query in relational algebraic form since for the set of statements we support, there is an equivalent representation in relational algebra [20]. As an example, the relational algebraic form of the constraint given in Figure 2 is shown as:

\[
(\sigma_{d.idSel:fr.read}(\sigma_{r.read}=200(\sigma_{r.id=d.idSel}(\sigma_{d.dist<10}(\text{RobotRadEv} \times \text{RobotDistEv})))))
\]

involving the relational operations selection (\(\sigma_F\)), projection (\(\pi_{d_1,d_2,...,d_m}\)), and Cartesian product (\(\times\)). The inner-most operation, Cartesian product is performed first, followed by the select operations from inner to outer, and concluding with projection. This relational query has a graphical representation as an abstract syntax tree as is shown in Figure 6.

According to [18], constraints specifiable with a subset of ATSQL2 (the subset of which our language is a proper subset), have a corresponding relational algebraic form involving the relational operations selection
(σ_F), projection (π_{d_1, d_2,..., d_m}), Cartesian product (×), and intersection (∩). But there is an important distinction between a query and its relational algebraic form. SQL allows a relation to have two or more tuples identical in all their attribute values. In general, then, unlike relational algebra where a relation is defined as a set of tuples, an SQL relation is a multiset, or bag, of tuples. This distinction, though important to database management systems, is trivialized in our on-line environment where every event is assigned a unique physical timestamp. The implication of this is that special operators are not required to remove duplicates.

It is assumed the constraint condition is in conjunctive normal form, that is, a select consists of simple comparisons connected only by OR’s and multiple selects are AND’ed together.

![Parse tree for C:1 before optimization](image)

Adapting operations to an event-based environment involves extensions which we discuss in the following sections.

**Selection** The selection operator is extended in a straightforward manner. A selection operator generates at most one output event for each input event. Storage is maintained for the temporal operators. A problem in implementing selection is in deciding how fully to implement the PRECEDES operator. The expression α PRECEDES β evaluates to true when an event β is received and there exists an event α in the history buffer such that the timestamp of α is earlier than the timestamp of β. To implement the PRECEDES operator fully requires infinite storage; an unrealistic option. We adopt the solution taken in [18] by introducing a *bounding parameter* to limit the life of an event participating in a select operation (and in the process to limit the semantics of the PRECEDES operator as well). With the user defined bounding parameter, selection will maintain the α event in its history buffer until α has aged beyond the time allowed by the bounding parameter. Aged events are discarded during the course of routine comparisons.

**Projection** The projection operator is extended in a straightforward manner. Each operator generates one output event for each input event, and no event storage is needed.

**Cartesian product** Cartesian product is a join with an implicit condition where the condition can be stated as follows: for two events α and β, the Cartesian product, α × β, is taken if the timestamp of α is equal to the timestamp of β. The operator is a binary operator that concatenates an event arriving on the left with all stored events on the right. Similarly, an event arriving on the right is concatenated with all stored event on the left. Thus, multiple output events are generated for each input event. The complexity
of Cartesian product arises in that it can be practically unbounded in both time and space; its efficiency is strongly dependent on the number of stored events from one input which are concatenated with the arriving event.

The implicit condition makes the problem more tractable by allowing representation with finite buffer space. But what is a reasonable buffer size? This depends strongly upon the temporal ordering of the two input streams and on their relative synchrony [18]. For two event streams, each with an imposed partial order and containing an event for every logical timestep, the storage requirements are minimal. But in the absence of one or the other, a suitable buffer size cannot be determined. For example, suppose a hazard condition occurring as two events, $\alpha$ and $\beta$, are generated from sources at some physical distance to one another, and large network delays postpone the arrival of $\alpha$. By the time $\alpha$ has arrived at the Cartesian product operator, $\beta$ has already been purged from the buffer and the violation is missed. Our solution to the buffer size problem is threefold: to impose a partial event ordering, to allow the user to specify buffer size, and to perform optimizations to reduce the number of events a Cartesian product must process. Partial event ordering could be applied to the event stream before events reached the dispatcher. A partial ordering tool has been developed at Georgia Tech as part of Falcon [8] and could be adapted to our environment with minimal effort. Optimization techniques reduce the need for larger buffers by applying select operations early to reduce the number of events reaching Cartesian product.

The join condition, comparing the timestamps of the two participating events, can be a physical time based on machine clock cycle or a logical time. If the former, it is likely that two events that 'occurred simultaneously' from an external user point of view, will fail on the join condition. To compensate for clock skew present in distributed systems, we introduce the notion of a safety-margin [13], an $\epsilon$ value that relaxes the definition of temporal equality. Safety-margin is a parameter to the system at startup.

To summarize, in adapting the operators to an on-line environment we have introduced three parameters: Cartesian product buffer size, safety margin, and a bounding parameter. Cartesian product buffer size specifies the size of the Cartesian product buffers. The safety margin, applied during Cartesian product, relaxes the rule used in determining whether two events 'occur at the same time'. The bounding parameter, applied during selection, is used to bound temporal operators by limiting the time interval that can exist between the occurrence of two temporally related events.

4.1 Static Optimizations

Query optimization is the application of heuristic rules to a query with the expectation that the resulting query is more efficient than the original version. We consider traditional optimizations (i.e., pushing selects and projects down in the abstract syntax tree) but also novel optimizations involving cost data and selection criteria, traditionally associated with query plan selection. The latter we include as dictated by the needs of Cnet.

Cost data is commonly used in relational databases during query plan selection to estimate the I/O cost of the query. I/O costs include access cost to secondary storage, storage cost, cost of storing intermediate files [7]. For Cnet's application, we do not care about I/O costs and instead formulate cost functions that are useful during query optimization.

The compiler applies optimization techniques after the query is parsed, a parse tree built, and an attribute list built for each node in the tree. We implement optimization itself as a multi-phase task where one heuristic rule is applied at each phase. In the first phase, select operations are pushed down the AST so they can be applied as early as possible in order to reduce the number of tuples appearing in Cartesian product operations. The second phase pushes projects down the tree in an effort to discard attributes needed by subsequent operations. Phase three considers cost data and is discussed in detail in [15]. Code generation is performed in the final phase.

Figures 7, 8, and 9 depict the algorithms used during phase ones through three, respectively. At each node in the parse tree (until a leaf is reached), there exists a parent and a child node. For every parent/child pair, an optimizeNode function is invoked depending on the type of the parent and type of the child. If, for example, the phase is one, the parent is a select node, and the child a project node, then the first function of Figure 7 is invoked. The heuristic applied when a parent select and a child project are encountered is to have the parent jump the child meaning the project will become the new parent and the select the new child. Where both parent and child are select nodes, the algorithm simply returns.
Whereas Figure 6 shows an unoptimized naive query, Figure 10 shows the parse tree after heuristic optimization has been applied. Once optimization is complete, a final pass is made over the parse tree to generate code. This final step is discussed next.

```c
optimizeNodes(..., SelectNode * parent, SelectNode * child);
   // do nothing

optimizeNodes(..., SelectNode * parent, ProjectNode * child);
   // select always jumps project

optimizeNodes(..., SelectNode * parent, CartesNode * child);
   // If select cannot be pushed down either side of cartes, then return
   // select node as parent. Otherwise split select and push parts down
   // sides or push all down one side if nothing to split and
   // return cartes as new parent.
```

Figure 7: Phase 1 optimization algorithms.

```c
optimizeNodes(..., ProjectNode * parent, ProjectNode * child);
   // Cascade of projects; all but last can be ignored. Return
   // child as new parent.

optimizeNodes(..., ProjectNode * parent, SelectNode * child);
   // Push project? Take union of attribute lists as new project list.
   // If project list equal to attribute list of select's source,
   // do not push. If project's list equal to select's list, remove
   // project. Otherwise push project.

optimizeNodes(..., ProjectNode * parent, CartesNode * child);
   // Push project below cartes, return cartes as new parent
```

Figure 8: Phase 2 optimization algorithms.

```c
optimizeNodes(..., SelectNode * parent, SelectNode * child) {
   if (parent->costFunction() < child->costFunction()) {
      push parent below child
      return child as new parent  // parent and child swap positions
   }
   return parent            // no change to parse tree
}
```

Figure 9: Optimization algorithm considering cost data

4.2 Code generation

Code generation results in a set of calls to the Cnet library in the form of Tcl commands. The commands are read by a Tcl interpreter embedded in the analysis executable and used to invoke functions in the Cnet library that build nodes and cnets.

Preceding and following traversal of the parse tree, the compiler generates commands to perform node setup. They include:

- create a node object,
- create a gate operation,
- link operations, and
- invoke node wrapup.

The node create command causes the library to allocate space for the node data structure; it returns a pointer to the structure that is used in subsequent calls to operations belonging to the node. The gate
operation command creates a gate operation that serves both as the entry point to the node and point-of-control for the node. Point-of-control activities include activating or deactivating the node, triggering cost updates, or deleting the node. Link commands cause the library to link two operations. Two operations are linked when the sending operation has a pointer to the input queue of the receiving operation. The wrap-up command causes the now established node to register its existence, and input and output needs with the dispatcher.

The command syntax using a subset of the commands generated for the constraint C:1 is illustrated in Figure 11. The sample script is divided into three sections (demarked by pound signs): a preface section, a body, and a conclusion. The preface creates a node and a single gate operation. The node is assigned a name (C:1); the name is then used throughout the script to reference the node. The gate create command accepts two event types: ROBOT\_STATE\_EV and ROBOT\_DIST\_EV and needs a single event queue, evq0, accepting events of type ANY\_EV.

The body creates three operations, a project, a Cartesian product, and a select. The code for each is demarked by a comment line beginning with a `#'. The project operation creates a ROBOT\_STATE\_EV event from attributes of a ROBOT\_STATE\_EV event. Its function is to strip attributes from an event thus making it smaller. Its declaration as an INTERNAL\_OP indicates that its output event is not to be considered a node level output event. It requires a single input queue, evq4.

The Cartesian product operation is created with two event queues accepting events of type ROBOT\_DIST\_EV and ROBOT\_STATE\_EV events, respectively. The select operation compares attributes of a ROBOT\_DIST\_EV with those of a ROBOT\_STATE\_EV event. The logical operator is equality (eq) and the attributes compared are the first attribute of both events (given by the 1 in the 'set op' line). Select is also an internal operation; its output is directed not to the outside world but to another operation (not shown). It requires a single queue that accepts events of type AGG\_EV, an internal event type that is a concatenation of events produced for example, by Cartesian product. The 'pick' name preface appearing in the commands name user-supplied functions that return an event attribute given an index value specifying the attribute's position. The 'assign' preface names functions used by projection to do the reverse: given a value and a position indicator, make the assignment.

The conclusion contains link and wrapup commands. Link commands require the name of the sending operation and of the receiving queue. The commands shown link the gate operation to the project and Cartesian product input queues, project to the second event queue of Cartesian product, and Cartesian product is to the select input queue. A full version of the script for C:1 is given in Appendix B.
Figure 11: Subset of script for constraint C:1.
5 Cnet Architecture and Implementation

A cnet is implemented as a set of nodes, with queues between nodes and a dispatcher controlling overall execution. A node in our terminology is an entity encapsulating all or part of a constraint. Nodes are connected in a DAG where an arc exists between two nodes $A$ and $B$ if $B$ requires an input event type generated by $A$. Nodes communicate via queues. If $A$ and $B$ are connected, when $A$ generates an output event, it is queued at $B$'s input queue. The dispatcher is responsible for event handling. We describe each in some detail below.

5.1 Monitoring Infrastructure

Cnet communicates with an application via a monitoring and communication infrastructure shown in Figure 12. The robotics simulation executes as multiple threads, with one robot per thread. At instrumented points, a thread invokes sensors that write event data to a shared buffer. Falcon [8], running as one or more separate threads, extracts event data from the buffer and sends it to the analysis tool in a binary encoded format, PBIO [5].

![Diagram](image)

Figure 12: The communication infrastructure.

Events arrive atDataExchange[6], a communication library serving primarily in this context as an event handler. DataExchange decodes events using PBIO before passing them on to the dispatcher. The dispatcher, as explained below, then passes events to those constraints requiring them. If a violation occurs in a constraint, and one of the accompanying actions is a steering command, the constraint generates a local event that is passed to the dispatcher. The dispatcher periodically services the local event queue, and invokes DataExchange to pass the steering command to Falcon, which decodes the event and invokes an appropriate steering function in the application code.

5.2 Event Handling and Analysis

The dispatcher’s primary function is in controlling the cnet’s execution. As such, it is responsible for:
- maintaining a record of the current set of active nodes,
- linking nodes, and
- maintaining information about the relationship between nodes.

More importantly, however, it acts as an event servicing agent, processing events from a number of sources:
- event data from application
- management (i.e., ADD, DELETE) requests from the constraint management interface

15
• constraint replace requests from the optimizer
• action requests from the nodes themselves.

Event flow to and from the dispatcher is depicted in Figure 13 where the shaded area represents tasks internal to the analysis tool. Event data from the application comprises the significant majority of all events processed by the dispatcher. Other event types, though occurring less frequently, are no less important. User requests originating in the constraint management interface take the form of requests to add or remove constraints from the system. Periodic triggering of dynamic query reoptimization may occur on the basis of run-time data accumulated during execution. The optimizer notifies the dispatcher when reoptimization has completed. It is the function of the dispatcher to guide actual replacement of the old constraint with the optimized version. Nodes themselves have actions to be executed when a constraint is violated; the dispatcher enacts the actions on behalf of nodes.

Figure 13: Event dispatching in the analysis tool.

5.2.1 Fairness in Rule Processing

In an environment so diverse, fairness in servicing events of different types is important. Equally as important, fairness must exist in servicing nodes. That is, no constraint should suffer starvation (constraint starvation exists when an event arrives for a constraint but the constraint either never receives the event or never gets activated to process the event.)

The rule processing algorithm shown in Figure 14 addresses fairness by a depth first search of the node graph. The graph is traversed whenever an event arrives. Branches of the tree are pruned under three conditions: the events fail to cause a constraint violation, the node is not interested in the event, or not all event types are present. Consequently, the depth search continues down a path only when the event is relevant and constraint violations occur. This significant pruning saves the approach from being wholly inefficient. A negative implication though is that a constraint violation will never occur for a hazard condition that manifests itself as the inability of a sensor to generate its event data. To deal with such conditions, one could employ a technique such as watchdog timers [9] to detect the unanticipated absence of sensor sources.

We describe the algorithm in the context of Figure 15. The net is implemented as a DAG with multiple entry points. The entry point arcs correspond to flows of primitive events (e.g., a, b, d, g), whereas the arcs between nodes correspond to flows of derived events, (e.g., c, e, f, h, i). Nodes when created register their input event needs with the dispatcher. Node C, for example, registers need for event type d while node G needs d and g. When an event d arrives, for instance, the dispatcher queues it at the input queues for
Figure 14: Rule processing algorithm.

6 Future Work

Continuous safety-critical systems are becoming common in modern society. Growing along with their proliferation is an awareness that existing approaches, often based on formal techniques, are not enough to provide the required assurance of safety. Needed is a complementary detection approach. Our research meets the needs of continuous safety-critical systems by providing a external, language-based approach to hazard detection. The language-based approach with its prescriptive language leaves much room for tailored optimizations, both static and statistical, or dynamic, in nature. The logically external approach encapsulates queries facilitating both distribution of analysis components and query reoptimization.

The principle contribution of this paper is the application of algorithms for statistics-based optimizations to achieve dynamic constraint reoptimization. The utility of optimization is demonstrated by experiments with a robotics application. Performance results indicate that the analysis tool supports efficient constraint analysis for reasonable sized constraint sets.

There are a number of promising ongoing and future research topics. Ongoing work includes distributing the analysis components to increase the scalability of the approach and to take advantage of available resources for analysis. We are also currently measuring the performance of the statistical optimization techniques and algorithms.

Future work includes expanding cost data and cost function heuristics to better utilize the dynamic flexibility built into the implementation, and the application of the analysis tool to a virtual environment. For instance, in the Iowa Driving Simulator (IDS) [12] a fully immersive ground-vehicle simulator can place a driver in a highly realistic driving environment. Hazard conditions in such an environment are the same as in real-life but without the attendant risk of harm or loss.

References


Figure 15: Sample Cnet.


A Syntax of the Cnet constraint language

This appendix presents the syntax of Cnet Constraint language in standard BNF. Non-terminals are enclosed by angular braces (“<””) and in slanted font. The empty term is represented by ε. Not shown are definitions of four trivial non-terminals: ⟨identifier⟩, ⟨character string⟩, ⟨integer⟩, and ⟨float⟩. An ⟨identifier⟩ is a string of letters, an underscore, or digital characters, starting with a letter or an underscore. A ⟨character string⟩ is a string of any printable characters. An ⟨integer⟩ is an integer constant and ⟨float⟩ a real constant.

⟨data-definition⟩ ::= ⟨create-rule⟩ | ⟨alter-rule⟩ | ⟨drop-rule⟩
   | ⟨activate-rule⟩ | ⟨deactivate-rule⟩
⟨create-rule⟩ ::= ⟨create-statement⟩ ⟨condition-statement⟩ ⟨action-statement⟩
⟨create-statement⟩ ::= CREATE RULE ⟨rule-name⟩ ON ⟨event-list⟩
⟨alter-rule⟩ ::= ⟨alter-statement⟩ ⟨condition-statement⟩ ⟨action-statement⟩
⟨alter-statement⟩ ::= ALTER RULE ⟨rule-name⟩ ON ⟨event-list⟩
⟨drop-rule⟩ ::= DROP RULE ⟨rule-name⟩ ON ⟨event-list⟩
⟨deactivate-rule⟩ ::= DEACTIVATE RULE ⟨rule-name⟩ ON ⟨event-list⟩
⟨activate-rule⟩ ::= ACTIVATE RULE ⟨rule-name⟩ ON ⟨event-list⟩
⟨event-list⟩ ::= ⟨event-name⟩ | ⟨event-name⟩, ⟨event-list⟩
⟨condition-statement⟩ ::= IF ⟨query-expression⟩
⟨action-statement⟩ ::= THEN ⟨action-list⟩
⟨action-list⟩ ::= ⟨action⟩ | ⟨action⟩ ⟨action-list⟩
⟨action⟩ ::= ACTIVATE ⟨rule-name⟩
   | DEACTIVATE ⟨rule-name⟩
   | STEER ⟨function-name⟩ ⟨argument-list⟩
   | EXEC_FUNC ⟨function-name⟩ ⟨argument-list⟩
⟨query-expression⟩ ::= ⟨select-exp⟩ | ⟨proj-exp⟩ | ⟨cartes-exp⟩
⟨select-exp⟩ ::= SEL ⟨term⟩, ⟨term⟩, ⟨binary-cond-list⟩
⟨proj-exp⟩ ::= PROJ ⟨term⟩, ⟨attribute-list⟩
⟨cartes-exp⟩ ::= CP ⟨term⟩, ⟨term⟩
⟨attribute-list⟩ ::= ⟨attribute⟩ | ⟨attribute⟩ ⟨attribute-list⟩
⟨nested-query⟩ ::= NOT EXISTS ⟨time-attribute⟩ ⟨query-expression⟩
⟨term⟩ ::= ⟨relation-name⟩ | ⟨query-expression⟩
⟨binary-cond-list⟩ ::= ⟨binary-cond⟩ | ⟨binary-cond⟩ OR ⟨binary-cond-list⟩
⟨binary-cond⟩ ::= ⟨scalar-val⟩ ⟨binary-op⟩ ⟨scalar-val⟩
⟨binary-op⟩ ::= < | <= | > | >= | = | <> MEETS | PRECEDES
   | PRECEDES ⟨time-attribute⟩
⟨time-attribute⟩ ::= IN ⟨time-val⟩
⟨argument-list⟩ ::= ⟨argument⟩ | ⟨argument⟩ ⟨argument-list⟩
⟨scalar-val⟩ ::= ⟨integer⟩ | ⟨float⟩
⟨rule-name⟩ ::= ⟨identifier⟩
⟨event-name⟩ ::= ⟨identifier⟩
⟨function-name⟩ ::= ⟨identifier⟩
⟨relation-name⟩ ::= ⟨identifier⟩
⟨attribute⟩ ::= ⟨identifier⟩
⟨argument⟩ ::= ⟨integer⟩ | ⟨float⟩
⟨time-val⟩ ::= ⟨integer⟩
⟨constant⟩ ::= ⟨integer⟩
B  Code Generation Script for C:1

Node_create  
C:1 C:1 disp0

Op_GATECreate
set param_list [list ROBOT_STATE_EV ROBOT_DIST_EV]  
gate0 C:1 GateC:1 2 $param_list  
EventQ_setProperties  
evq0 gate0 gateQC:1 ANY_EV

# pj(d.idSelf, d.dist)  
Op_PjCreate
set param_list [list ROBOT_DIST_EV pickRobotDist 1 1 \  
ROBOT_DIST_EV pickRobotDist 3 2 ]  
proj0 C:1 ProjC:1 ROBOT_DIST_EV assignRobotDist 3 4 \  
2 $param_list INTERNAL_OP  
EventQ_setProperties  
evq1 proj0 proj0Q0C1 ROBOT_DIST_EV

# sel(d.dist < 10)  
Op_SelCreate
set op1 [list EVENT ROBOT_DIST_EV 3]  
set op2 [list STR_CONST NO_EV 10]  
sel0 C:1 SelC:1 $op1 pickRobotDist $op2 null \  
lt_i ROBOT_DIST_EV INTERNAL.OP  
EventQ_setProperties  
evq2 sel0 sel0Q0C1 ROBOT_DIST_EV

# pj(d.idSelf)  
Op_PjCreate
set param_list [list ROBOT_DIST_EV pickRobotDist 1 1 ]  
proj1 C:1 ProjC:1 ROBOT_DIST_EV assignRobotDist 2 3 \  
1 $param_list INTERNAL.OP  
EventQ_setProperties  
evq3 proj1 proj1Q0C:1 ROBOT_DIST1_EV

# pj(r.rad, r.id)  
Op_PjCreate
set param_list [list ROBOT_STATE_EV pickRobotState 1 1 \  
ROBOT_STATE_EV pickRobotState 2 2 ]  
proj2 C:1 Proj2C:1 ROBOT_STATE_EV assignRobotState 3 4 \  
2 $param_list INTERNAL.OP  
EventQ_setProperties  
evq4 proj2 proj2Q0C:1 ROBOT_STATE_EV
# jn()
0p_CartesCreate
cartes0 C:1 cartes0C:1
EventQ_setProperties
evq5 cartes0 cartes0Q0C:1 ROBOT_DIST_EV
EventQ_setProperties
evq6 cartes0 cartes0Q1C:1 ROBOT_STATE_EV

# sel(r.id = d.idSelf)
0p_SelCreate
set op1 [1 list EVENT ROBOT_DIST_EV 1]
set op2 [1 list EVENT ROBOT_STATE_EV 1]
.sel1 C:1 sel1C:1 $op1 pickRobotDist $op2 pickRobotState \ 
  eq_i AGG_EV INTERNAL_OP
EventQ_setProperties
evq7 sel1 sel1Q0C:1 AGG_EV

# sel(r.rad >= 200)
0p_SelCreate
set op1 [2 list EVENT ROBOT_STATE_EV 2]
set op2 [1 list STR_CONST NO_EV 200]
.sel2 C:2 sel2C:2 $op1 pickRobotState $op2 null \ 
  geq_i ROBOT_STATE_EV OUT_OP
EventQ_setProperties
evq8 sel2 sel2Q0C:1 AGG_EV

0p_LinkCreate
link0 gate0 evq1
link0 gate0 evq4
link0 proj0 evq2
link0 sel0 evq3
link0 proj1 evq5
link0 proj2 evq6
link0 cartes0 evq7
link0 sel1 evq8

Node_wrapUp C:1 disp0