Software Approach to Hazard Detection Using On-line Analysis of Safety Constraints

Beth Schroeder*
Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{beths,schwan}@cc.gatech.edu

Sudhir Aggarwal
State Univ. of New York
at Binghamton
Binghamton, NY 13902
sudhir@cs.binghamton.edu

Abstract

Hazard situations in safety-critical systems are typically complex, so there is a need for means to detect complex hazards and react in a timely and meaningful way. This paper addresses the problem of hazard detection through the development of an on-line analysis tool. The approach allows the user to specify complex multi-source hazards using a query-like language, uses both synchronous and asynchronous on-line checking approaches to balance efficiency and expressiveness, accommodates dynamic applications through dynamic constraint addition, and supports distributed and parallel applications running in heterogeneous environments.

1 Introduction

Public awareness of safety issues involving computers is growing as incidents resulting in loss of life or near loss receive wide publicity. One of the more tragic examples involved the Therac-25, a therapeutic linear accelerator. Six people either died or suffered serious injuries from massive overdoses of radiation between 1985 and 1987 before the problems were acknowledged and corrected [11]. A *safe system* is one that is free from accidents or unacceptable losses [17]. There is important research and development being done in providing intrinsically safe systems [8, 15, 16, 20, 21], systems incapable of evolving into a state that could lead to injury or loss of life. But the goal of an intrinsically safe system is difficult to achieve for some of today's complex, dynamic applications running in parallel or distributed environments. Adopting guidelines established by system safety engineers [7], if one cannot guarantee an intrinsically safe system, the next preferred approach is a technique that prevents, minimizes, or detects the presence of hazards. A *hazard* is a state or condition of the system that combined with some environmental conditions can lead to an accident or loss event [17]. Automatic pressure relief valves, lockins, lockouts, and interlocks are common hardware hazard prevention approaches. An example of a software approach is a trip computer in a nuclear power plant that initiates procedures to shutdown the plant when operating conditions are hazardous [12].

The research here addresses the problem of enhancing software safety through hazard detection. The premise of our work is that hazard situations can and do occur, and are often complex, involving multiple sources. So there is a need for a mechanism to detect complex, multi-source hazards and react in a timely and meaningful way. This paper addresses such a detection mechanism through Cnet, an on-line analysis tool that supports the specification of complex hazards using a query-like language, uses both synchronous and asynchronous checking approaches to balance efficiency and expressiveness, accommodates dynamic applications through dynamic constraint addition, and supports distributed and parallel applications running in heterogeneous environments.

We have applied our detection approach to a set of autonomous robots that can navigate toward a goal across unmapped terrain, `reacting' to stimuli in the environment. Envisioning more relaxed definitions of hazards, our detection approach is useful in many environments. For instance, in the Iowa Driving Simulator (IDS) [10] 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. For example, excessive driving speeds on icy roads create hazard conditions to be avoided in either a virtual or real world. Hazard detection could

*This work was funded in part by NSF equipment grants CDA-9501637, CDA-9422033, and ECS-9411846.

profitably be applied during development of a virtual reality driving simulator, allowing developers to play what-if scenarios by iteratively adjusting speed and environmental conditions to determine what combination of conditions will be classified a hazard to which the simulator will react.

1.1 Problem and Solution Strategy

Hazard detection is a viable approach to improving software safety [12] and, in some cases, is the only approach. For example, the confluence of events that caused the 1990 AT&T system runaway (a combination of heavy load, software errors, and neighboring switches) caused a 9 hour nationwide blockade that was not anticipated despite extensive testing of the involved 114 electronic switching systems [18]. Multiple cause failures such as this can be most easily mitigated with detection oriented approaches. Our goal is to provide such a detection approach through on-line hazard analysis.

Our approach consists of a language for specifying complex hazards, a library for building executable versions of the hazard descriptions, and a run-time environment to handle execution. In particular, hazards are represented as constraints specified on application behavior. These constraints, called *safety constraints* are specified with a rule language. The compiler generates a list of commands to the library which builds an executable entity, or node, for each constraint as a collection of selection, projection, and join operations. The nodes are linked together at runtime as a directed acyclic graph (DAG). As shown in Figure 1, the compiler accepts sensor and constraint specifications. From a sensor specification it generates sensor definitions that are used to instrument the application code. Information about sensor definitions is also used in calls to the library. The safety constraint specifications are compiled into a sequence of Tcl [19] commands which are executed by the Tcl interpreter resident in the executable. The interpreter is used initially when the graph is being created and thereafter only when a constraint is dynamically added to Cnet, so the performance impact generally associated with interpreters is minimized. The Dispatcher is responsible for event handling, event distribution, action execution, and user command handling. The user intefrace provides a vehicle through which commands to add, modify, and remove constraints are issued.

The most significant contribution of this work is its novel approach to hazard detection, highlighted by:

- accommodates dynamic applications through dynamic constraint addition, enabling, and disabling;
- provides synchronous and asynchronous constraint checking with predictable performance and low perturbation; and

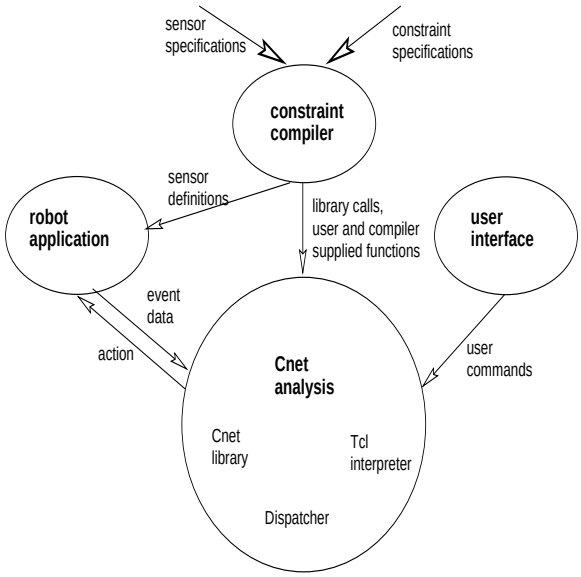

Figure 1. The system architecture includes the constraint compiler, instrumented application, Cnet analysis task, and user interface.

- supports specification of complex, multi-source hazards.

In contrast to previous static analysis approaches, Cnet accommodates dynamic applications by allowing dynamic constraint addition. Additional constraints can be specified over a newly created application process, task, or object as long as the application entity is instrumented.

Synchronous and asynchronous constraint checking is provided by flexible placement of constraints. This allows constraints with localized sensor data needs to be evaluated in the sensor itself. Predictable performance guarantees can be made because of the DAG restriction on the constraint graph. Low perturbation is achieved through the use of a separate thread to perform monitoring and synchronous constraint checking. Perturbation could be further reduced by employing a technique used in software fault isolation [26] of embedding monitoring related instructions in open slots in the instruction stream.

A multi-source hazard is a hazard that occurs when a combination of conditions exist: a failed valve indicator, condensate pump out of order, and a relief valve failing to close, for example. It is likely that each condition occurs in a separate piece of controlling software. To enforce such a hazard, the analysis tool must be located such that it can gather information from each source. This precludes a completely synchronous approach. Additionally, the more complex the hazard, the stronger the need for a language in which to express complex relationships between conditions

of a hazard and between hazards themselves.

There is ample justification for implementing the analysis tool outside the application. The external nature of the tool makes possible the specification of complex constraints. It also makes possible its use with legacy systems with only minimal modifications needed for instrumentation. Also, by isolating safety related code we not only reduce the perturbation of one or more application components, but by separating the safety related code from the application, we provide a well defined interface between the two that minimizes the risk of the safety checking code introducing hazards itself. Separate constraint analysis means more portable constraint analysis as well. Our analysis tool can be used with any system providing a uniform event stream.

Our focus is largely pragmatic. Our first concern has been to identify and support a variety of useful abstractions. Over time we will determine the limits of the approach and the degree to which reliance on the approach is justifiable as we gain experience applying the tool to varied applications.

The remainder of the paper is organized as follows. In Section 2 we the example application used in our work and specify a couple sample constraints. In Section 3 we introduce the rule language while in Section 4 we introduce the library and run-time environment. Related work is discussed in Section 5. We conclude in Section 6 with a discussion of current status and future work.

2 Autonomous Robots Example

The application used in our work is a multiagent reactive robotic system simulation by Balch and Arkin [1] in their work on the Autonomous Robot Architecture (AuRA) at the Georgia Institute of Technology. The work was undertaken to investigate the importance of communication in robotic societies. The authors tested their strategy through an iteration of simulation and instantiation on real systems. Our work with robots is in the simulated environment with each robot running as a separate thread and employing shared memory as a communication medium.

We briefly describe the robots to give the reader sufficient background for understanding the constraints we specify below. The robots perform one of three tasks: forage, consume, and wander. During *forage*, a robot wanders in an environment, looking for attractors. Upon encountering an attractor, it moves toward the attractor, attaches itself, and returns the object to a specified home base. During *consume*, a robot wanders the environment, looking for attractors. Unlike forage, after attachment a robot performs work on the object in place. 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. The robots are 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 to back wander.

There are meaningful constraints that can be specified over the behavior of the group of robots just described. For instance, suppose a group of robots are tasked with digging up drums filled with radioactive material that are buried over a large 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 be greatly concerned 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 danger of picking up debris from the contaminated robot does not occur until the approaching robot is within 5 ft. With this constraint in place and given a work area large enough and the number of attractors great enough, the user may be confident enough to allow work to continue even after a robot becomes contaminated. The constraint ensures the user will be notified when a robot has been contaminated, and will be notified again, with enough time to respond, when another robot has entered the `danger zone' . The danger zone constraint could trigger an alert to the operators to halt the approaching robot or it could invoke a user-defined handler in the approaching robot to change its state so it no longer seeks to assist the contaminated robot. The constraint for the scenario just described is as follows:

- If the radioactivity level of a robot exceeds 200 roentgens per hour, then a violation occurs if a robot approaches within 10 ft. of the radiating robot more than once.

The constraint specifies that if any robot becomes radiated, a violation occurs if any other robot comes within 10 ft. of the radiated robot. A second sample constraint has a management focus: gathering statistical information for evaluating performance:

- 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, a violation has occurred.

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.

Why not simply modify the robot application to enforce the constraint behavior? Embedding constraint checking in the robot application suffers the same limitations as other embedded approaches: first, a single robot knows at most state and goal information about another robot. Global, statistical knowledge cannot be known or computed by any one robot. Second, constraints that dynamically modify parameters such as attraction/repulsion forces could be added to the application but at the cost of recompiling and relinking the application code. In a fluid environment where constraint specification is itself a dynamic process, subject to learning on behalf of the users, evolving and dynamic applications, and evolving environments, the additional functionality provided by Cnet is best implemented separate from the application. In the next section we discuss the rule language used for specifying constraints.

3 The Rule Language

The constraints given in the robot example are expressed in a natural language. A natural language is obviously not a suitable choice for specifying constraints that are to transformed into executable entities - particularly when done dynamically. A query language, on the other hand, given its declarative style and ease of use, is suitable. The language we have adopted for our use is based on the active database rule definition language of the Starburst system [27]. The language consists of five commands: **create rule**, **alter rule**, **drop rule**, **activate rule**, and **deactivate rule**. The **create rule** is used to define a new constraint. The syntax of the command is:

create rule *name* **on** *event-type*
if *condition*
then *action-list*

The *name* names the rule and each rule is defined on a set of event-type. The **if** clause specifies the rule's condition. The condition is the constraint that is checked when the rule is triggered by an incoming event. Our language uses the temporal query language ATSQL2 for specifying the condition. ATSQL2 is a variant of TSQL2 [24] and is currently being proposed for incorporation into SQL3 [25]. The condition can be any valid ATSQL2 query. The **then** clause specifies the rule's actions. An action is executed when the rule is triggered and its condition is true. Actions are discussed in Section 3.2.

Figure 2 illustrates a **create rule** command. The command creates a rule named *C:1*. The event sources needed

```
CREATE RULE C:1 ON robotRad, robotState
IF
   SELECT radiatedRobot r.ID, r.rad
   FROM robotRad as r, robotState as s
   WHERE
      s.task = FORAGE and s.state = DELIVER and
      r.rad >= 200 R and r.ID = s.ID
THEN
   STEER disableRobot r.ID
```

Figure 2. Notify if foraging robot in `deliver' state has radioactivity level that exceeds 200 roentgens per hour.

by this rule are *robotRad* and *robotState*. Event sources may originate from a sensor in the application or as the result of another query. The **if** statement delineates the rule's condition. The condition can be any ATSQL2 query. In the example, the query is composed of a SELECT statement, a FROM statement, and a WHERE statement. The SELECT statement builds a new relation or event type from attributes of one or more existing event types. The derived event, radiatedRobot, will contain three attributes: a robot ID and roentgen level taken from the robotRad relation and a timestamp derived from the timestamps of the tuples satisfying the condition. The FROM statement defines variable names *r* and *s* that will represent the event types *robotRad* and *robotState*, respectively. The WHERE clause specifies a predicate on the explicit attributes that selects those events that will contribute toward the new event type. The **then** statement delineates the action list. In this example there is a single action, a *STEER* command. The first parameter to *STEER* names a function in the robot application to be invoked. The second parameter identifies the task to be affected by the action.

3.1 Applying a Query Language to On-line Monitoring.

Snodgrass has shown that a relational database query language can be used beneficially to specify queries that are evaluated against event streams such as are generated with on-line monitoring [23]. The significant difference between evaluating queries against a database and evaluating them against an event stream is that in the latter constraints must be evaluated against a *conceptual* database rather than an *actual* database. That is, no database *per se* exists. Instead, each constraint must have sufficient storage to maintain the application state it needs. Hence, a given event may exist in multiple nodes at any moment; the length of time an event remains in the node depends on the attributes upon which the query is based and the complexity of the query.

There is an issue of efficiency with which one must deal when executing queries in an on-line analysis environment. Instead of a query being executed periodically or upon user demand, and a set of tuples satisfying the query returned, the query is in essence executed every time an event arrives. What keeps this characteristic from being wholly inefficient is that a constraint by its nature will reject the majority of the events it receives. To further enhance efficiency, we have in place compile-time optimization techniques to order the operations such that event discarding occurs as early in the sequence of operations as possible. Additionally, work on run-time optimization is underway.

3.2 Action Statements

The user has control over the set of executing constraints in two ways. The first is through issuing commands (e.g., **alter rule**) through the user interface. The second is through a rule's action statement. An action statement is a command listed in the action part of a rule. The set of allowable actions must be basic enough and broad enough such that when taken alone or combined, they allow the user to effect a desired behavior. Three action statements are supported:

- invoke a user-defined function in node;
- invoke a handler in the application; and
- enable or disable a constraint.

Invoking a user-defined function in a node may do something simple like causing a bell to ring or a message to be printed to an operator console or something more complex like collecting statistics. Invoking a handler in the application causes a steering command to be issued to the application that results in the execution of a user-defined function residing in the application. The mechanisms for such steering are discussed in the next section.

Enabling and Disabling Constraints. When a constraint is disabled, it no longer processes events although its code is still resident in the system. Enabling, the default mode, reverses the disabling action and allows the constraint to process events once again. Constraint enabling and disabling has multiple useful application.

Any approach, such as the one discussed in this paper, that allows dynamic constraint addition must deal with the eventuality that an added constraint will conflict with an existing one. The effect of such 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. To obviate the problem, the DISABLE clause is provided as a means for the user to manage the conflict.

The enable and disable clauses are also 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 the hope that the obstacle will move. If this simple error recovery fails, its second response would be to determine a new route.

4 Library and Dispatcher

The rule language provides a means for specifying constraints. The library and run-time environment, on the other hand, provide the means for the specified constraints to be transformed into individual executable entities and the mechanism to execute the constraints against the incoming event stream. The **library** is a collection of functions that build two types of compnents: operations and nodes. An *operation* is a component that implements one of selection, projection, or join. Control flows between operations by procedure calls. A *node* is a collection of operations with a number of methods defined for it. A node can be created and can accept connections. Additionally, it possesses general information about itself so it can respond to questions as to its state (active, inactive), it can be called upon to activate or deactivate itself, or it can return a list of its input events and output events. Control flows between nodes under the control of the dispatcher.

The **dispatcher** controls net execution. At startup, it awaits nodes to register their existence and event needs. The dispatcher links those nodes together having data dependencies as shown by their event lists. During execution, the Dispatcher accepts events from the application and routes them to the interested nodes and accepts and executes commands from the user.

4.1 From Rules to Executable Entities

The transformation of a constraint from a rule to a node begins with the rule compiler. The rule compiler parses a constraint, and converts it to a relational algebraic expression in conjunctive normal form. From the relational algebraic expression an abstract syntax tree is constructed and it is from this that the optimizer performs compile-time optimization before generating a sequence of Tcl commands [22]. So roughly for every select, project, and cartesian product operation in the relational algebraic expression, there is a corresponding Tcl command in the script file. Also included in the script file are Tcl commands to build the node, the entity to which the operations belong, and to link the operations in the proper order.

The initial set of queries become executable nodes when the analysis tool is first executed. The dispatcher accepts the name of a script file as an argument, and invokes the Tcl interpreter, passing it the name of the script file. The interpreter executes the commands in the file, each command resulting in a call to the library. Through a sequence of calls the node is built. The script file can contain any number of

queries in any order. Linking queries takes place when a node registers itself with the dispatcher.

4.2 Multi-Source Hazards

Multisource hazards, hazards which can be described as consisting of events from multiple sources, make up an important and substantial subset of hazard descriptions [12]. Implementing detection of such hazards requires making tradeoffs between latency, perturbation, and ease of use. Latency is far more of an issue in external approaches to hazard detection than it is, for example, when constraints are embedded directly in the application code. But embedded approaches suffer from increased perturbation and decreased breadth of potential event sources. Our external approach trades decreased perturbation and the ability to specify multi-source hazards for increased latency. More complex hazard descriptions also require a more general language, such as a query language, to distinguish events from mulitple sources and describe complex relationships between events in a natural way.

To achieve efficient communication between multiple sources and the analysis tool, we employ a communication infrastructure, DataExchange [5], developed at Georgia Tech. DataExchange provides for binary IO of event data between the multiple sources and the analysis tool. The full features of DataExchange, in its ability to forward data to multiple clients based on event type, can be utilized in the version of Cnet underway where the analysis tool is itself distributed.

4.3 Dynamic Applications

Support for dynamic applications occurs in part by the design concept of *event types*. All events possessing the same set of attributes belong to the same event type, regardless of the sensor from which they originate. For example, every event consisting of a robot ID, location, and current timestamp is of the *robotID* event type, regardless of which of the many robots generated the event. A constraint node registers its interest in event types, so any dynamically created application task generating events of a known event type will automatically be included in the constraint checking done by nodes accepting events of that type.

Support for dynamic applications based solely on event types would only partially solve the problem. Also needed is the ability to add new constraints. With constraints descriptions encoded as Tcl scripts, dynamic constraint addition becomes straightforward. As shown in Figure 3, when an *ADD* command arrives at the dispatcher (from the user interface), the dispatcher invokes the Tcl interpreter, passing it the script file name as an argument. The interpreter executes the script, the execution of which results in a series of calls to the Cnet library to build a node. The new node, once built, automatically registers its existence with the dispatcher, providing it a list of input and output events of which the dispatcher adds to its internal lists. Control then returns to the dispatcher and event processing resumes. Events arriving at the dispatcher for which the new node is interested will immediately be forwarded to the new node.

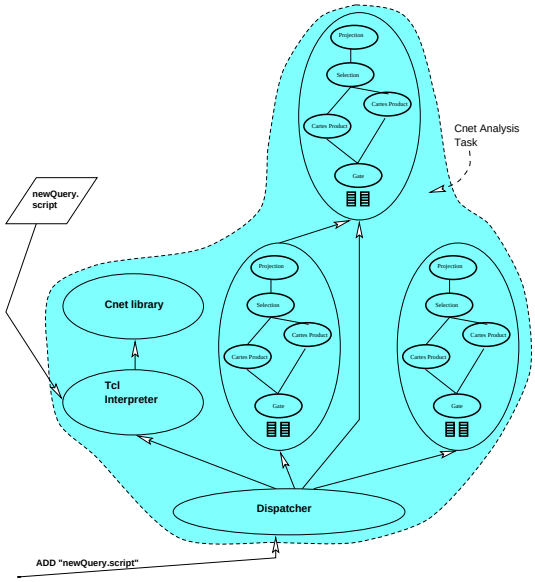

Figure 3. The Cnet analysis task includes a dispatcher, Tcl interpreter, library, and constraint nodes.

4.4 Synchronous Constraint Checking

Asynchronous constraint checking has been explained in some detail in the general discussions of Cnet. Less has been said, however, about synchronous constraint checking. Synchronous constraint checking is checking performed in the application data space. It is suitable for filtering data to reduce the volume of events flowing to the analysis tool. Synchronous placement is restricted to constraints having a single input event type. The restriction is necessary because the constraint node is invoked from within the sensor generating the event type. The constraint is evaluated prior to the event being forwarded to the analysis tool.

The constraint compiler provides support for handling synchronous constraint checking by identifying those constraints suitable for synchronous placement. A Tcl interpreter creates the synchronous nodes in a separate thread in the application data space. Since we do not want the overhead of the interpreter executing while the application is running, we restrict dynamic constraints to an asynchronous implementation.

5 Related Work

Parallels to our work can be drawn from a couple of distinct bodies of work: assertion checking in real-time systems and applying query languages to on-line monitoring.

Assertion Checking in Real-Time Systems. Gerber's work [6] on guaranteeing end-to-end timing constraints is an automated design methodology that generates a solution for a set of tasks that keeps consistent a set of end-to-end timing constraints. Gerber's work is a prevention approach with the goal of designing an architecture such that it is impossible for any of the constraints to be violated. Ours is a detection oriented approach: we provide a mechanism for detecting constraint violations that we maintain cannot all be known a priori. The work is important to us though in that it defines a class of constraints specifiable and monitorable by our tool. The Monitoring and Assertion tool (MAC) [2] is a formal analysis technique for monitoring symbolic execution traces generated by the Modechart Toolset [3]. It provides a mechanism for evaluating properties of the system on a particular execution trace. Leveson's work in the early 1980's [13] is early recognition of the need for run-time checking for hazard prevention. Her synchronous approach, though, requires embedding constraints in the application.

On-line Application of Query Languages. In Liu and Pu's work on continual queries [14], a client specifies continual queries over information stored in a distributed interoperable environment such as the Internet. The objective is to compute the query and return the entire resulting relation upon the first triggering only. On subsequent triggerings, only the add, modify, and delete change information is returned. The primary intent is to minimize the amount of information returned as a result of the query and is most effective when the returned relation is large. The immediate response requirements of our environment, however, force us to reevaluate a query every time relevant event data arrives (or in database terms, every time a change takes place to one of the underlying relations) so the returned relation is small; often a single tuple. The underlying assumptions of the two approaches are quite different, hence continual queries cannot be applied to our problem.

It has been shown that the relational model is an appropriate formalism for the information processed by the monitor though earlier applications of this formalism to monitoring were primarily for performance evaluation [9, 23]. In addition, prior approaches to monitoring were static, that is, they required that all constraints be known at compile time. Given the exploratory and `what-if' potential of safety constraints, any realistic solution must allow for dynamically added constraints.

6 Conclusion

The research presented here addresses the problem of improving software safety through hazard detection. The approach consists of a query-like language and compiler for specifying hazards, a library for creating operations and nodes, and a run-time tool to dispatch arriving events, trigger node execution, and handle dynamic node addition.

Hazards are often described by a number of events occurring simultaneously. Any realistic approach to hazard detection must accommodate the complex hazard as easily as it accommodates the simple one. Our general language approach allows for the specification of complex constraints specified over distributed components. Just as hazards are complex, applications are growing more complex. It is not unusual to find real-time applications characterized by the dynamic addition of objects or tasks. Though these applications do not lend themselves well to formal analysis techniques, they are still amenable to hazard detection approaches, particularly ones that allow constraints to be added dynamically.

For a hazard detection approach to be realistic, however, it must be responsive. That is, it must recognize a hazard and respond within a reasonable period of time. That period of time is the latency. Part of our effort to minimize latency is concentrated on employing both synchronous and asynchronous constraint checking with the emphasis on using synchronous checking to filter event data.

There are several avenues of long term pursuit. We would like to apply hazard detection to a virtual environment, where hazards still have meaning though in a less critical context. Additionally, hazard descriptions often include components outside the software system. A hazard detection approach should be able to accommodate a description that includes state from these components as well. One approach is to mirror these non-software components by adding shadow objects representing the component. These shadow objects would then be the source of state information for the analysis tool. Finally, the purpose of hardware hazard detection is often as a safeguard. It is the mechanism, such as a pressure relief valve, to which an engineer turns for the extra measure of safety. Such a device must at all times provide the extra measure of safety, not decrease the overall safety of the device on which it is placed. We need to explore and delineate the situations in which our software approach can be justifiably used in hazard detection. This final piece of work is one of its most important.

References

[1] Tucker Balch and Ronald Arkin. Communication in reactive multiagent robot systems. *Autonomous Robots*, 1(1):27–52, 1994.

[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] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose. MT: A toolset for specifying and analyzing real-time systems. In *Proceedings IEEE Real-Time Systems Symposium*, December 1993.

[4] Ingeman J. Cox and Narian H. Gehani. Exception handling in robotics. *Computer*, 22(3):43–49, March 1989.

[5] Greg Eisenhauer, Beth Schroeder, Karsten Schwan, Vernard Martin, and Jeffrey Vetter. DataExchange: High performance communication in distributed laboratories. In *Accepted for publication in 9th Int'l Conference on Parallel and Distributed Computing and Systems*, October 1997.

[6] Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In *Proceedings 15th Real Time Systems Symposium*, pages 192–203. IEEE, December 1994.

[7] W. Hammer. *Handbook of system and product safety*. Prentice Hall, 1972.

[8] M. P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency checking of software requirements. *IEEE Transactions on Software Engineering*, 22(6), June 1996.

[9] Carol Kilpatrick, Karsten Schwan, and David Ogle. Using languages for capture, analysis and display of performance information for parallel and distributed applications. In *Proceedings 1990 Int'l Conference on Programming Languages*, 1990.

[10] Joh Kuhl, Douglas Evans, Yiannis Papelis, Richard Romano, and Ginger Watson. The Iowa Driving Simulator: An immersive research environment. *Computer*, 28(7):35–41, July 1995.

[11] Nancy Leveson. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[12] Nancy G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, pages 34–46, February 1991.

[13] 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.

[14] Ling Liu, Calton Pu, Roger Barga, and Tong Zhou. Differential evaluation of continual queries. Technical Report TR95-17, Department of Computer Science, University of Alberta, 1996.

[15] Robyn R. Lutz. Targeting safety related errors during software requirements analysis. In *Proceedings 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, 1993.

[16] Louise E. Moser and P.M. Melliar-Smith. Formal verification of safety-critical systems. *Software - Practice and Experience*, 20(8):799–821, August 1990.

[17] title = Nancy G. Leveson, editor.

[18] Peter G. Neumann. *Computer Related Risks*. Addison-Wesley, 1995.

[19] John Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, 1995.

[20] Vivek Ratan, Kurt Partridge, Jon Reese, and Nancy G. Leveson. Safety analysis tools for requirements specifications. In *Proceedings Compass96*, June 1996.

[21] Joh Damon Reese and Nancy G. Leveson. Software deviation analysis: A “safeware” technique. Technical report, University of Washington, 1996.

[22] Beth Schroeder, Sudhir Aggarwal, and Karsten Schwan. Hazard detection using on-line analysis of safety constraints. Technical Report GIT-CC-97-01, College of Computing, Georgia Institute of Technology, 1997. http://www.cc.gatech.edu/tech_reports.

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

[24] Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.

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

[26] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings 14th SOSP*, pages 175–188, December 1993.

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