A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools

Nelly Delgado, *Student Member*, *IEEE*, Ann Quiroz Gates, *Member*, *IEEE Computer Society*, and Steve Roach, *Member*, *IEEE Computer Society*

Abstract—A goal of runtime software-fault monitoring is to observe software behavior to determine whether it complies with its intended behavior. Monitoring allows one to analyze and recover from detected faults, providing additional defense against catastrophic failure. Although runtime monitoring has been in use for over 30 years, there is renewed interest in its application to fault detection and recovery, largely because of the increasing complexity and ubiquitous nature of software systems. This paper presents a taxonomy that developers and researchers can use to analyze and differentiate recent developments in runtime software fault-monitoring approaches. The taxonomy categorizes the various runtime monitoring research by classifying the elements that are considered essential for building a monitoring system, i.e., the specification language used to define properties; the monitoring mechanism that oversees the program's execution; and the event handler that captures and communicates monitoring results. After describing the taxonomy, the paper presents the classification of the software-fault monitoring systems described in the literature.

Index Terms—Assertion checkers, runtime monitors, specification, specification language, survey, software/program verification.

1 Introduction

Runtime software monitoring has been used for profiling, performance analysis, software optimization as well as software fault-detection, diagnosis, and recovery. Software-fault detection provides evidence that program behavior complies or does not comply with specified properties during program execution. While other verification techniques, such as testing, model checking, and theorem proving, aim to ensure universal correctness of programs, the intention of runtime software-fault monitoring is to determine whether the *current* execution preserves specified properties; thus, monitoring can be used to provide additional defense against catastrophic failure and to support testing by exposing state information. The increasing complexity and ubiquitous nature of software systems and the cost and inadequacy of testing [64] has sparked renewed interest in the field.

Following the standard usage [2], we define a *software failure* to be a deviation between the observed behavior and the required behavior of a software system. A *fault* occurs during the execution of software and results in an incorrect state that may or may not lead to a failure. An *error* is a mistake made by a human that leads to a fault that may result in a failure.

Many tools have been proposed for runtime monitoring with the purpose of detecting, diagnosing, and recovering from software faults. The most recent comprehensive survey of the field of runtime software monitoring was published in 1981 by Plattner and Nievergelt [59]. This work examined concepts, goals, and limitations of monitors. Unlike their survey and other work that considers application-specific monitors [30], [31], [68], [75], the taxonomy presented in this paper provides a classification based on the application and implementation of monitors that are used for software-fault detection, diagnosis, and recovery [16]. The taxonomy is used to classify a representative sample of the diverse range of runtime monitoring tools. It categorizes the various runtime monitoring research in a manner that supports identification of

1. classes of software properties that can be specified and applied,
2. required infrastructure,
3. support provided to the user, and
4. types of applications that are targeted by a given monitor.

In addition, the work provides a common language for discussing runtime monitoring systems and their components.

2 Scope and Definitions

Software requirements are implementation-independent descriptions of the external behavior of a computation. They answer the question: What behaviors of the software are acceptable? *Software properties* are relations within and among states of a computation. Equivalently, software properties can be defined as a set of sequences of states [1]. They answer the question: What relations about states of a computation lead to acceptable external behavior? A monitor uses properties to discover faults prior to their becoming failures. A *specification language* is a language

- *N. Delgado is with Microsoft, Bellevue, WA 98007. E-mail: ndelgado@cs.utep.edu.*
- *A.Q. Gates and S. Roach are with the Department of Computer Science, The University of Texas at El Paso, El Paso, TX 79902. E-mail: {agates, sroach}@cs.utep.edu.*

Manuscript received 23 Dec. 2003; revised 13 Aug. 2004; accepted 10 Sept. 2004.
Recommended for acceptance by R. Lutz.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0209-1203.

used to specify requirements of the problem domain and other properties associated with software behavior.

An executing program P has a set of threads that may have sequential, interleaved, or parallel execution [72]. The state Σ_P of an executing program P is the state of the store and the individual threads executing in that program. The *execution trace of a program P* is a sequence, possibly infinite, of program states.

There are various definitions for runtime software-fault monitors (henceforth referred to as monitors) in the literature. The definition most in agreement with the interpretation considered in this paper is "A *monitor* is a system that observes the behavior of a system and determines if it is consistent with a given specification" [55]. A monitor takes an executing software system and a specification of software properties and checks that the execution meets the properties, i.e., that the properties hold for the given execution. The work presented here focuses only on monitors that are used to detect faults. Other monitoring systems extend this capability by diagnosing faults, i.e., providing information to the user that will aid the user in understanding the cause of the fault and assist the system in recovering from faults by directing the system to a correct state (forward recovery) or by reverting to a state known to be correct (backward recovery) [3].

Monitoring is concerned with actual transitions between states, not possible transitions. A monitor takes an execution trace and a software property specification and checks that the execution trace meets the property, i.e., that the property holds for the given trace. A monitor typically attempts to verify the property while the software system is executing. A software property often has the form $\mu \rightarrow \alpha$, where μ is some condition on $\Sigma_{\mathrm{P}}^{\mathrm{i}}$ that identifies the states in which α must hold. This means that in any state where μ is *true* in $\Sigma_{\mathrm{P}}^{\mathrm{i}}$, then α must also be *true* in $\Sigma_{\mathrm{P}}^{\mathrm{i}}$. If α evaluates to *false*, then the current execution has reached a disallowed state. It is possible that the specification does not restrict the states in which α must hold, in which case, μ is *true*, e.g., in the case of an invariant.

In practice, it is possible that the current state $\Sigma_{\mathrm{P}}^{\mathrm{i}}$ does not contain enough information to evaluate $\mu \rightarrow \alpha$. For example, suppose a property written for an expression evaluator states that the number of left parentheses must never exceed the number of right parentheses in the input stream. The constraint might be specified $\top \rightarrow (N_{right} \leq N_{left})$, where N_{right} and N_{left} are counts of the number of right and left parentheses encountered in the input stream up to the current state. The program, however, may not explicitly record N_{right} and N_{left}. This information may be obtained by scanning through all of the previous states; however, this would require the monitor to store all of the information for every state during an execution. A more practical solution is for the monitor to a priori consider the information that will be needed to evaluate μ and α (N_{right} and N_{left} in this example) and make this information an explicit part of the state of either the executing program or the monitor.

As shown in Fig. 1, a monitor is composed of two parts: an *observer* that evaluates μ and an *analyzer* that evaluates α. For example, suppose a program must never allow two processes access to a critical region simultaneously. This property may be specified for a monitor as: "When a process enters the critical region, the number of processes in the critical region must be exactly one." The observer checks the program state to determine if a process has entered the critical region. When it detects this event, the analyzer tests the proposition that the number of processes in the critical region is exactly one.

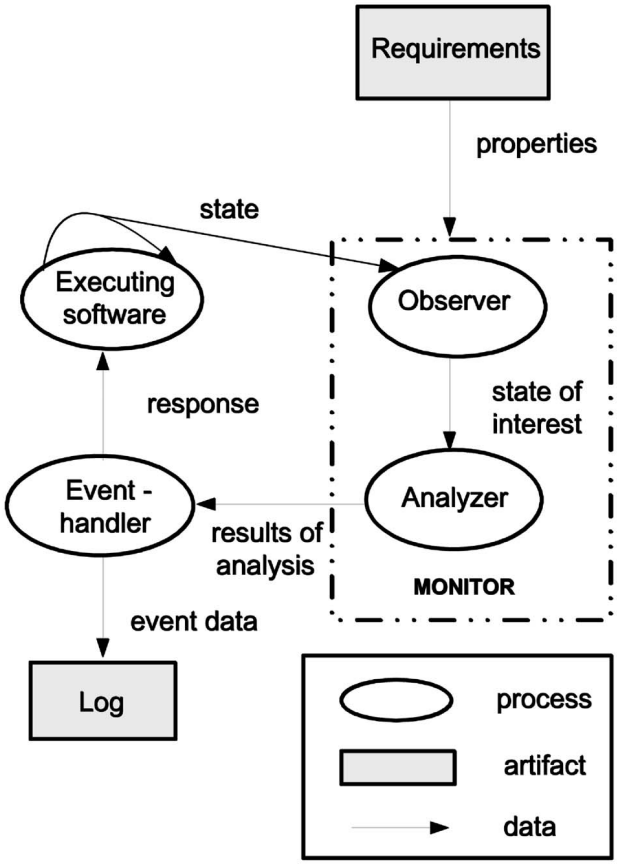

Fig. 1. High-level view of a runtime monitor.

When a violation of a property is detected by the analyzer, the monitoring system must respond in some fashion. The response could require the system to initiate an action such as halting the program, entering a recovery routine, or sending event data to a log. The *event-handler* is the mechanism that captures and communicates the monitoring results to the system or user and possibly responds to a violation.

3 Runtime Monitoring Taxonomy

A survey of the current literature on runtime monitoring identified a wide spectrum of tools that have monitoring capabilities. The literature has widely divergent nomenclature, and we have made an effort to use the most common words and meanings. This section describes the taxonomy of runtime software-fault monitoring systems. The intent of the taxonomy is to allow a user to understand and classify the state-of-the-art in monitoring approaches and techniques, provide a basis for discussion about these tools, identify classes of software properties that can be specified and applied to a given system, and identify existing tools.

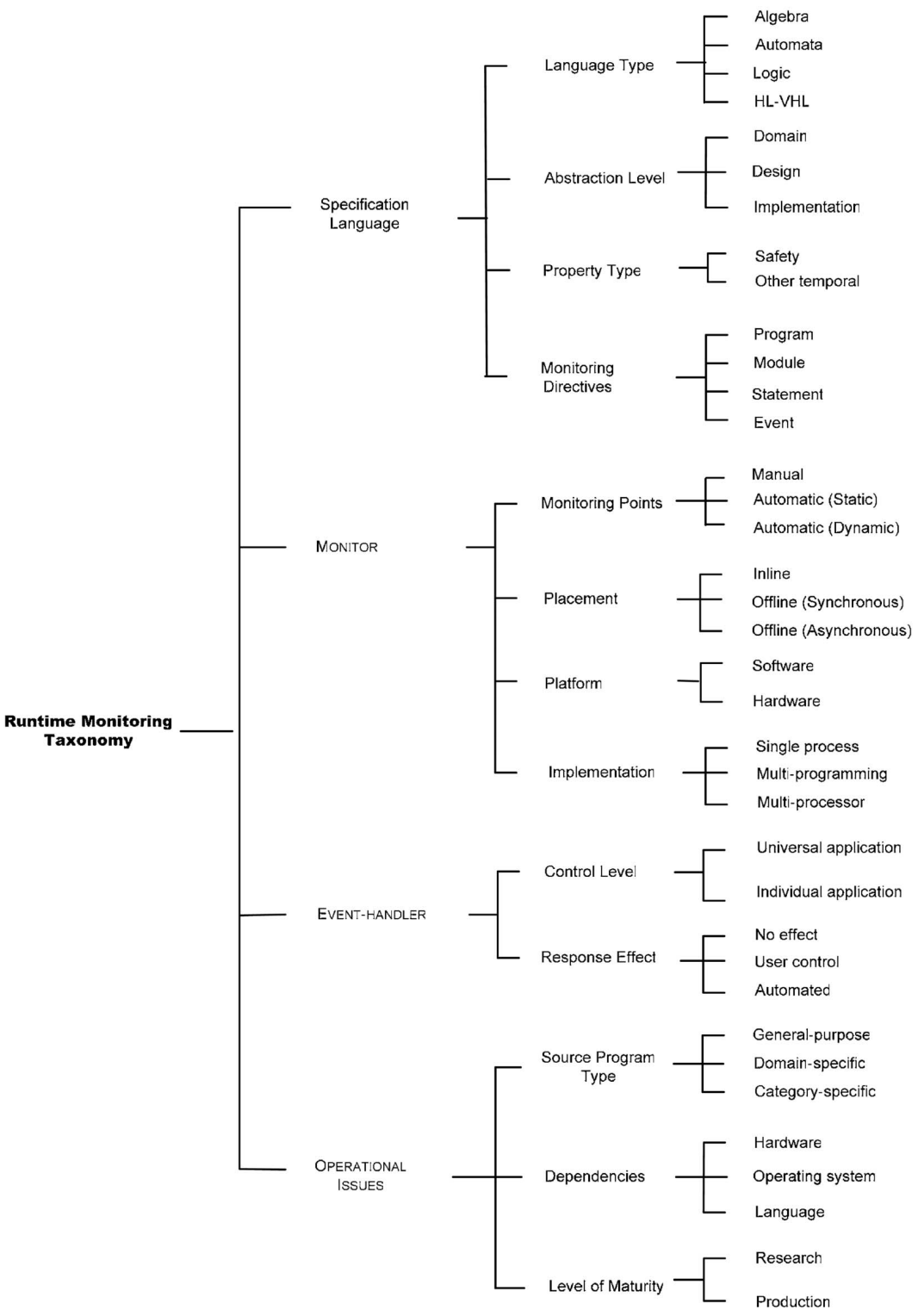

Fig. 2. Runtime monitoring taxonomy.

To meet these needs, we based the taxonomy on common elements of monitoring systems: specification language, monitor, and event-handler. In addition to these elements, the taxonomy considers operational issues, such as the type of programs targeted by the monitoring system, platform dependencies, and level of maturity of the tool. Fig. 2 shows the taxonomy.

3.1 Specification Language

The *Specification-Language* branch of the taxonomy as shown in Fig. 2 classifies the language that is used in a tool to define monitored properties, the abstraction level of the specification, and the expressiveness of the language (property type and level of monitoring).

Language Type. The language used to specify a property can be based on algebra, automata, or logic. In this case, the language is classified accordingly. The HL/VHL category denotes that the specification language is a functional, object-oriented, or imperative language or it is an extension of the source language.

Abstraction Level. Abstraction level refers to the support that the language provides for specifying properties and knowledge about the domain, design, or implementation. A language that provides constructs to support the expression of properties in a particular domain or supports capture of knowledge about the problem domain is classified as *domain-based*. For example, a language that provides templates for specifying properties of Web services would be classified as domain-based. *Design-based* properties include specifications concerning software design at a level that is independent of implementation. For example, pre and postconditions on methods are considered design-based properties. A language that supports specification of properties about the implementation of a program is classified as *implementation-based*, e.g., properties directly related to programming constructs or statements. For example, if the language allows one to specify a property concerning the range of a variable or the boundaries of an array, then the monitor would be classified as implementation-based.

Property Type. There are two types of properties considered: safety and other temporal. A *safety* property [5] expresses that something (bad) never occurs. Safety properties include, for example, invariants, properties that define a sequence of events, properties that check values of variables, and properties that deal with resource allocation. The *other temporal* category includes properties such as progress and bounded liveness as well as timing properties. This category is not a disjoint set with safety, i.e., the properties in some cases are instances of safety properties. For example, deadlock-freeness could be classified as a safety property that states that the system can never be in a situation in which no progress is possible. In addition, bounded liveness, e.g., "event A occurs within 10 seconds," could be classified as a safety property. An example of a timing property is: "event A must occur before event B."

Monitoring Directives. A property can be evaluated at different levels within a program: program, module, statement, and event. Program-level properties are those that are specified on threads or on relations between threads. Examples include program invariants that hold across all state changes in a program, assertions on global data structures, and interactions between modules. Module-level properties are those that are specified on functions, procedures, methods, or components, such as abstract data types or classes. Examples of module-level properties include class invariants and pre or postconditions on a procedure or method. Statement-level properties are those that are specified for a particular statement. Event-level properties are those that are defined based on a state change or sequence of state changes. For example, the property "every update to variable x in class C is with a positive number" is an event-level property if the monitoring system can detect a change in x and initiate the appropriate check. If the specifier has to associate the property with each statement that modifies x, it is considered a statement-level property.

3.2 Monitor

A monitor observes and analyzes the state of the system. The monitor checks correctness by comparing an observed state of the system with an expected state of the system. The event-handler acts on the results from the monitor, if any. There are numerous approaches for implementing monitors. The distinguishing characteristics summarized in Fig. 2 are: monitoring points, placement, platform, and implementation.

Monitoring Points. Points in the program at which execution of monitoring code will be initiated are referred to as *monitoring points*. Classification of the monitoring points can be manual or automatic. *Manual* classification includes manual instrumentation of the program's source code, intermediate code, or object code. *Automatic* classification includes tools that can automatically detect points of instrumentation. If analysis is required to determine monitoring points, it can be achieved through *dynamic* or *static* analysis, or a combination of both, in which case both are marked. For example, a monitoring system that uses a control-flow graph to assist in identifying points of instrumentation would be classified as automatic (static). Another example is a system that uses pre and postconditions to specify properties upon entry and exit of a module. Those that require evaluation of the states during runtime or evaluation of an execution trace would be classified as automatic (dynamic).

Placement. Placement refers to where the monitoring code executes. The monitoring code can perform an *inline* check, i.e., the monitoring code is embedded in the target code (including calls to subroutines). In this case, the monitor uses the resources of the monitored program. On the other hand, the monitoring can be performed *offline*, meaning that the monitor executes as a separate thread or process, possibly on a separate machine. If the application can continue to execute without waiting for the analyzer to complete or if the monitor uses an execution trace that is analyzed after execution, then the monitor is classified as *offline* (*asynchronous*). If the application must wait until the check is complete, then the monitor is classified as *offline* (*synchronous*) in the taxonomy. Monitors that are initially offline (asynchronous), but use synchronization points, e.g., checkpoints, are classified as offline (synchronous).

Platform. Monitors have commonly been differentiated on the basis of software and hardware monitors [75]. A *software* monitor uses code to observe and analyze the values of monitored variables. A *hardware* monitor may be a microprocessor attached to a system with connectors or a hardware device connected to the buses of the target system that allow it to detect events of interest or collect relevant data [75].

Implementation. There are three ways in which a monitor can execute. The *single process* classification indicates that the monitor executes in the same process as the target program. The *multiprogramming* classification indicates that the monitor and target program are executing as separate processes or threads on the same processor. In the *multiprocessor*

classification, the monitor and target program execute on different processors. The latter two dimensions further classify offline in the placement classification.

3.3 Event-Handler

Monitors observe behavior of programs and can react to a violation of a specification in a particular state. The event-handler refers to how the monitor reacts. Response actions can alter the application state space, report application behavior, or start up another process.

Control Level. The control level classification separates those monitors that react universally to a violation from those that permit the user to individually specify the actions of the monitor or program as a result of a violation. If the event-handler is only capable of giving a single response to any event, e.g., a system can only log a constraint violation, the tool is classified as *universal application*. *Individual application* indicates that the user is able to specify different responses to different events.

Response Effect. This category reflects the extent to which the monitor's response to a violation can affect program behavior. Examples of monitors that have *no effect* on program behavior (except for execution speed and program size) are those that report violations or create a trace. Examples of systems that are classified as *user control* are those that use breakpoints or other facilities that require user interaction with the system when a violation is detected. Examples of *automated* systems are those that use graceful degradation, termination, or recovery actions, such as program steering, exception handling, or memory rollback to respond to violations. Program steering is defined as the capacity to control the execution of a program. See Gu et al. [30] for a review of research in dynamic and interactive program steering.

3.4 Operational Issues

In choosing monitoring tools, there are overarching considerations that deal more with the external environment rather than with the monitor itself. The classification includes source program type, dependencies, and level of maturity.

Source Program Type. Source type describes the type of programs to which the monitoring tool is applied. The classifications include general-purpose, domain-specific, and category-specific. *General-purpose* indicates that the monitoring system targets programs written in general-purpose programming languages and does not rely on domain-specific information other than that contained in the specified properties. *Domain-specific* indicates that the monitoring system targets a specific environment and the monitoring system contains knowledge about the domain that is used, e.g., nuclear reactors or air traffic control. *Category-specific* indicates that the monitoring system targets a specific class of programs, e.g., real-time systems or distributed systems.

Dependencies. Dependencies capture restrictions on the monitoring tool. Some monitors can only operate on certain hardware, operating systems, or programs written in a specific language.

Level of Maturity. Level of maturity rates the stage of development of the monitoring tools. This category differentiates between research prototypes and tools available for public use either as a commercial product or via a maintained website.

4 Classification of Monitoring Systems

The tools classified through the taxonomy were identified through a review of the literature. This work focuses on runtime software-fault monitors as defined earlier. Tools such as profilers and debugging tools and systems used to detect hardware failures are excluded.

Tables 1, 2, and 3 provide the breakdown of the tools with respect to the specification language, monitor, and event-handler, and operational-issues branches of the taxonomy, respectively. The tools have been classified by identifying the appropriate leaf nodes of the taxonomy shown in Fig. 2. More than one marking on a classification indicates that the tool is hybrid with respect to that classification.

5 Synopsis of Monitors

This section presents a brief summary of the software-fault monitoring systems that were described in the literature. When appropriate, the summary provides explanation of the classification.

5.1 Alamo

A Lightweight Architecture for Monitoring (Alamo) [38], [39], [40], developed for C and Icon programs, uses the Icon programming language to specify assertions. The Alamo monitoring architecture utilizes CCI, a Configurable C Instrumentation tool [74], as a preprocessor that uses parse trees to identify monitoring points and inserts events into the target program's source code. Alamo provides the facility to identify events at specific locations, allowing it to check properties at the statement as well as the expression level. The Execution Monitor (EM) executes the Target Program (TP) and then returns control to the EM with information in the form of an event report. The EM can query the TP for additional information, such as the values of program variables and keywords. The user can apply a predicate to each event report to make monitoring more specific, sample execution behavior not reported by events, or view detailed information through Alamo's visualization mechanism.

5.2 Annalyzer

The Annalyzer [48], [49] uses Anna (ANNotated Ada), an extension of Ada, to specify properties as annotations. The annotations are converted into code that checks the properties at runtime. The Annalyzer compares the runtime behavior of an application with properties that are specified at different levels of abstraction, e.g., package, subprogram, data structures, or statement level. Type or subtype annotations apply the property throughout the type definition's scope; subprogram annotations support specification of pre and postconditions. The monitor is used to pinpoint and analyze errors by using a "two-dimensional pin pointing" approach in which the Annalyzer initially finds errors using properties specified at a high level of

TABLE 1
Monitors Classified According to the Specification Language Branch of the Taxonomy

Tool	Language Type				Abs. Level			Property Type		Monitoring Directives			
	Algebra	Automata	Logic	HL/VHL	Domain	Design	Impl.	Safety	Other Temporal	Program	Module	Statement	Event
Alamo				×			×	×				×	×
Annalyzer				×		×	×	×		×	×	×	
Anna CCS				×		×	×	×		×	×	×	
APP				×		×	×	×			×	×	
BEE++				×			×	×					×
DB Rover			×		×	×	×	×	×	×	×	×	×
DynaMICs			×				×	×					×
Falcon				×	×		×	×				×	
Jass	×			×	×	×	×	×	×	×	×	×	×
JPaX	×		×				×	×	×			×	×
JRTM			×		×		×		×				×
MaC			×	×	×		×	×	×		×	×	×
MoP	×	×	×	×	×	×	×	×	×	×	×	×	×
Non-inter				×			×	×	×		×	×	
PMMS			×			×	×	×		×		×	×
RAC			×	×		×	×	×			×	×	
ReqMon			×		×		×	×	×	×	×		
Sentry			×	×			×	×	×	×		×	
T Rover			×		×	×	×	×	×	×	×	×	×

abstraction, i.e., at the package level, and then augments the program with more detailed properties at the subprogram and statement level. The event handler notifies the user when a violation occurs. An interface provides the user with options, including suppression of annotations, display of the executing program, and display of the program text around where the violation occurred.

5.3 Anna Consistency Checking System

The Anna Consistency Checking System (Anna CCS) [67] allows users to annotate an Ada program with properties written in Anna, an Ada extension for specifying properties. Anna CCS generates a checking function corresponding to each subtype annotation. A subtype annotation defines a property that must hold throughout the scope of the type definition. Calls to the checking functions are inserted at places where inconsistencies with respect to the annotation can arise (e.g., assignment statements, procedure-call statements, and type conversions). Anna CCS uses checking tasks that perform consistency checks concurrently with the execution of the underlying program. When encountering annotation inconsistencies, the event-handler can ignore the inconsistencies, report the inconsistencies, or terminate the program based on the specification.

5.4 Annotation PreProcessor (APP)

Annotation PreProcessor (APP) [62] translates an annotated C program into an equivalent C program with embedded assertion checks. An APP assertion specifies a property that applies to some state of a computation. The focus is on assertions on function interfaces and bodies. APP recognizes four assertion constructs: assume (specifies a precondition on a function), promise (specifies a postcondition on a function), return (specifies a constraint on the return value of a function), and assert (specifies a constraint on an intermediate state of a function). The programmer manually inserts assertions at points in the program where checks should occur. The compiler preprocessor pass is used to instrument the application. Assume, promise, and return are module-level monitoring directives. Assert is a statement-level directive. The programmer has the option to attach a severity level and specify a response to violated constraints. The severity level indicates the relative importance of an assertion and determines whether or not the assertion will be checked at runtime. APP supports specification of a response to a violation.

5.5 BEE++

BEE++ [7] is an object-oriented application framework for the dynamic analysis of distributed programs written in C or C++. It views execution of a distributed program as a stream of events. The monitor, referred to as an event interpreter, supports specification of high-level events from low-level events by way of inheritance. A sensor provides a placeholder for an event that is either user-defined or predefined by BEE++. When a sensor is encountered, or

TABLE 2
Monitors Classified According to the Monitoring Branch of the Taxonomy

Tool	Monitoring Points			Placement			Implementation			Platform	
	Manual	Automatic, static	Automatic, dynamic	In line	Off line, synchronous	Offline, asynchronous	Single processor	Multi-programming	Multi-processor	Software	Hardware
Alamo		×		×	×			×		×	
Annalyzer	×	×		×			×			×	
Anna ccs	×	×		×	×		×		×	×	
APP	×	×		×			×			×	
BEE++	×	×			×	×		×	×	×	
DB Rover		×	×		×	×			×	×	
DynaMICs		×		×	×		×		×	×	
Falcon	×					×	×	×	×	×	
Jass	×	×		×			×			×	
JPaX		×				×		×	×	×	
JRTM	×	×			×			×	×	×	
MaC		×	×	×	×	×		×	×	×	
MoP	×	×		×	×	×		×			
Non-Inter	×					×			×		×
PMMS		×		×			×			×	
RAC		×		×			×			×	
ReqMon		×			×			×	×	×	
Sentry	×	×				×		×		×	
T Rover	×	×		×			×	×	×	×	

triggered, runtime data is loaded into the event and it is sent off to each analysis tool that has bound itself to that sensor. The insertion of sensors in the application is the task of the programmer. The design is based on a symmetric peer-to-peer architecture with the ability to dynamically configure target applications and monitoring tools. B++ supports the ability to pause, restart, or kill an application and supports monitoring multiple clients at once, or monitoring a client with multiple views.

5.6 DB-Rover

The DB-Rover [18], [21] targets integrity validation of databases through the specification of temporal rules written in Linear-Time Temporal Logic (LTL) and Metric Temporal Logic (MTL). MTL properties can specify lower bounds, upper bounds, and ranges for relative-time and real-time properties. Through a GUI, users are able to describe LTL rules graphically and specify properties at all levels of the program. Graphical simulators enable simulation of the rules before deployment. The rules are translated automatically using Temoral Rover and then compiled. In this tool, the monitoring code is not embedded in the code, but resides on a standalone DB-Rover server that consists of a Web server and a validation host process. Triggers, such as a read operation or creation of a new record, activate the Web server and validation host process. The Web server receives information from the database and propagates the information to the validation host process. The validation host process checks the information against the temporal rules. DB-Rover can operate synchronously or asynchronously. It is also possible to capture data in a database and use DB-Rover to analyze the data at a later time. In DB-Rover, each specification requirement or rule has its own set of actions that are programmable by the user. The actions associated with success or failure can specify different behaviors. The applications that have been targeted include business and security applications.

5.7 Dynamic Monitoring with Integrity Constraints

Dynamic Monitoring with Integrity Constraints (DynaMICs) [27], [28] is a software tool that supports the elicitation of properties from domain experts, designers, and developers as well as their application to monitoring. Properties, expressed in logic, capture relations on objects modeled by the application, design limitations, and implementation assumptions. These properties are specified as event-condition-action, where the event defines the trigger for the check, the condition specifies the check, and the action defines the response to a violation.

The approach automatically inserts constraint-checking and knowledge-generating code by analysis of a program's path expression, which is generated from the control-flow graph of the program's intermediate object code. Monitoring code automatically generated from specifications

TABLE 3
Monitors Classified According to the Event-Handler and Operational Issues Branches of the Taxonomy

Tool	Control level		Response-Effect			Source Program Type			Dependencies			Level of Maturity	
	Universal Application	Individual Application	No Effect	User Control	Automated	General-purpose	Domain-Specific	Category-Specific	Hardware	Operating System	Language	Research	Production
Alamo		×		×		×				×	×		×
Annalyzer	×			×		×					×	×	
Anna CCS		×			×	×					×	×	
APP		×			×	×					×		×
BEE++		×		×				×	×	×	×	×	
DB Rover		×			×	×	×	×			×		×
DynaMICs		×		×	×	×					×	×	
Falcon		×			×			×			×	×	
Jass		×			×	×					×		×
JPaX	×		×			×					×	×	
JRTM		×	×					×			×	×	
MaC		×			×	×					×		×
MoP		×			×	×					×	×	
Non-inter	×		×					×	×		×	×	
PMMS		×			×	×			×			×	
RAC	×				×	×					×	×	
ReqMon	×		×				×					×	
Sentry		×			×			×			×	×	
T Rover		×			×	×	×	×			×		×

triggers the execution of monitoring code by another process or processor. Checking code resides in a separate module and is accessed by procedural calls or can be concurrently checked by a separate process or processor [73]. The event handler logs all violations, but the specification language allows an action to be specified to indicate the steps to be taken on a constraint violation. Other user support [26] includes the ability to display a violated property, related properties, supporting documentation, and code section when a violation occurs. DynaMICs is being designed for C, C++, and Java programs.

5.8 Falcon

Falcon [24], [29] provides online monitoring and steering of large-scale parallel programs. A monitoring specification consists of low-level *sensor specification constructs* and higher-level *view specification constructs*. Using an IDL-like language, programmers define application-specific sensors and probes for capturing the program and performance behaviors during runtime and the program attributes that direct steering. Probes update program attributes asynchronously to the program's execution. Actuators may also execute additional functions to ensure that modifications of program state do not violate program correctness criteria. Monitoring is accomplished through the use of runtime libraries for information capture, collection, filtering, and analysis.

Falcon's steering system permits users to implement online control systems that operate on and in conjunction with the programs being steered. The primary task of each steering server is to read incoming monitoring events and then decide what actions to take based on previously encoded decision routines and actions. Local monitoring and steering threads perform trace data collection, processing, and steering concurrently and asynchronously with the target application's execution. Local monitors and steering controllers typically execute on the target program's machine, but they may run concurrently on different processors by using a buffer-based mechanism for communication between the application and monitoring threads. An alternative approach performs all monitoring activities in the user's code.

Falcon is an implemented, category-specific monitoring system designed for distributed systems running under the Mach operating system. Implementation relies on the Cthreads library and can be run on several hardware platforms. Other systems related to Falcon include the Mirror Object Steering System [25] and ECho [23], a publish-subscribe communication system with capabilities that have much broader application.

5.9 Java with Assertions

Java with Assertions (Jass) [4] is a general-purpose monitoring approach that is implemented for sequential, concurrent, and reactive systems written in Java. A precomplier translates annotations to programs written in Java into pure Java code. Compliance with the specified annotations is dynamically tested during runtime. Assertions extend the

Design by Contract approach [50] that allows specification of assertions in the form of method pre and postconditions, class invariants, loop invariants, and additional checks to be inserted at any part of the program code. The choice and parallelism operators provide event monitoring directives. In addition to Boolean expressions, Jass allows programmers to use universal and existential quantifications that range over finite sets. Trace assertions, based on the process algebra CSP, describe the observable behavior of a class and are used to monitor the correct invocation of a method as well as the order and timing of method invocations. The generated code will check that the trace of the current program execution is included in the traces of the main process; otherwise, a trace exception is thrown that can trigger a user-defined rescue block. Refinement checks supports design by facilitating specification of classes on different levels of abstraction. Interference checks make it possible to detect when an assertion in one thread may become invalid through statements in another thread.

5.10 Java PathExplorer

Java PathExplorer (JPaX) [33], [34], [35], [36] is a general-purpose monitoring approach for sequential and concurrent Java programs. This tool facilitates logic-based monitoring and error pattern analysis. Formal requirement specifications are written in a linear temporal logic (both future time and past time) [63] or in the algebraic specification language Maude [12], [13]. JPaX instruments Java byte code to transmit a stream of relevant events to the observation module that performs two kinds of analysis: logic-based monitoring (checking events against high-level requirements specifications) and error pattern analysis (searching for low-level programming errors). Maude's rewriting engine is used to compare the execution trace to the specifications. Error pattern runtime analysis explores an execution trace to detect potential errors, including data races and deadlocks, even if these errors do not explicitly occur in the trace. JPaX can identify violations of bounded liveness properties.

5.11 Java Runtime Timing-Constraint Monitor

Java Runtime Timing-constraint Monitor (JRTM) [52], [53] targets timing properties of distributed, real-time systems written in Java. The timing properties express assertions that relate the time of occurrences of different events to one another in a specification language based on Real Time Logic (RTL). Java programmers insert the event triggering method calls in their Java programs where event instances occur. JRTM can detect some property violations statically through constraint graphs. Whenever an event is triggered at runtime, the application sends a message to the monitor reporting the occurrence time of the event instance and the event name. The monitor keeps these event occurrence messages in a sorted queue with the earliest event message at the head of the queue. The constraints are checked for violation and synchronization is enforced. The monitor can run on the same machine with a target process or on a stand alone monitoring machine.

5.12 Monitoring and Checking

Monitoring and Checking (MaC) [41], [42], [43], [44], [45], [46], [70] provides a framework for runtime monitoring of real-time systems written in Java. Requirement specifications are written in MEDL (Meta Event Definition Language). MEDL allows the user to define auxiliary variables that store values that can be used to identify events and conditions as well as guarded statements that can assign values to auxiliary variables for encoding information on past states. A condition is a state predicate, i.e., a proposition that is evaluated at each state of the computation, and an event is an instantaneous state change, i.e., a change of state in a condition from one value to another. MaC can monitor safety, bounded liveness, and other temporal properties.

A monitoring script, written by the user in PEDL (Primitive Event Definition Language), is used to monitor objects and methods. PEDL can look at local and global variables and can detect alias variable names. A filter maintains a table that contains names of monitored variables and addresses of corresponding objects. It acts as an observer that communicates the information that is to be checked by the runtime monitor. Monitoring points are inserted automatically since the monitoring script specifies which information needs to be extracted, not where in the code extraction should occur. Static analysis is used to determine monitoring points and dynamic analysis is used to reduce monitoring overhead by not evaluating new program state. The event-handler can emit a signal, map events, create a trace, or steer the program based on violation of specified conditions and actions. The user specifies steering conditions.

While monitoring and checking are done concurrently, there is no coupling between MaC's monitor and checker. The monitor can forward the output stream into a file and the checker can analyze it later. The event recognizer and filter are separate processes, although MaC components do have to run on the same processor.

5.13 Monitoring-Oriented Programming

Monitoring-oriented programming (MoP) [9] is an approach that allows formal property specifications to be added on top of a target programming language and to generate monitoring code from the formal specification. No restriction is placed on the formalism in which one chooses to write the specification as long as a translator exists. The generated code must contain the following components: declarations, initialization, monitoring body, success condition, and violation condition. The user annotates the points in the program at which monitoring code should be inserted. The annotation also contains information about the logic used. At the precompilation stage, the code generators take formal specifications and create monitoring code that is written in the same target language as the implementation. MOP supports multiple formalisms. It has been implemented for past time and future time linear temporal logic as well as extended regular expressions. A Java-MOP prototype has been implemented as a client-serve application. MOP can handle violations by executing

recovery code, displaying or sending messages, or throwing exceptions. It supports inline, offline synchronous, and offline asynchronous monitoring.

5.14 Noninterference Monitoring Architecture

The noninterference monitoring architecture [75], [76] monitors the execution of distributed real-time systems without interfering with their execution by using additional hardware to collect state information and assist in monitoring. The monitoring system consists of a set of monitoring nodes that is separate from the target system's communication network. Each monitoring node contains a hardware device to snoop the data, address, and control buses of the target node for specified conditions. The monitoring system does not use any of the resources of the target system. Monitoring can be done at the process-level (interprocess communication and synchronization), function-level, and instruction-level. The architecture is based on the bus structure of the MC68000 processor. The target language is restricted to block-structured programming languages where the scope of variables is determined statically and a process consists of a set of functions or procedures.

The data to be captured for each type of monitoring is predetermined. Process-level monitoring includes process creation, termination, synchronization, I/O operation, interprocess communication, wait child process, external interrupt, and process state change. Function-level monitoring includes function call and function return. Process-level data processing includes logical behavior and timing behavior. Timing analysis includes computation, scheduling, and synchronization (distributed termination, deadlock, starvation, and missed operation) problems. Timing constraints are classified into interprocessor timing constraints and intraprocessor timing constraints. Event-handling is not done at runtime; interpretation of collected data and timing analysis is done postprocessing.

5.15 Program Monitoring and Measuring System

Program Monitoring and Measuring System (PMMS) [47] is an approach that automatically collects information about the execution characteristics of a program. The user can specify objects and relations at the program, event, or statement-level. Primitive specifications describe execution activities of a program such as the following: operations on the data structures, execution of the control constructs, and temporal relationships among selected primitives. The user can specify preconditions, code to collect data, postconditions, and actions.

PMMS accepts the original program and properties given in a formal specification language. It installs instrumentation code in the program, determines what data must be collected, and inserts code to collect and process that data. The implementation uses temporal dependency among events to filter out irrelevant data at runtime and uses a main memory active database to facilitate the collection, computation, and access to results. PMMS handles events by installing code that reacts whenever relevant events occur.

5.16 Runtime Assertion Checker for the Java Modeling Language

The Runtime Assertion Checker for the Java Modeling Language [8], [10] assists in the generation of test oracles and identifies errors during testing of Java programs. Programmers write pre and postconditions for class methods using a side-effect-free subset of the Java Modeling Language. It is also possible to check class-level poststate assertions such as invariants. These annotations are translated into Java code embedded in the source code. When violations of pre or postconditions occur, exceptions are thrown. The runtime assertion checker keeps track of the location information of assertions, including the file name and location, in order to produce informative error messages.

5.17 ReqMon

ReqMon [60], [61] provides a framework and tool for monitoring requirements at runtime. In this framework, high-level requirements are translated into a Unified Modeling Language (UML) design model from which a Java program is generated. ReqMon generates a Promela model from the program source code and a monitored program event log from instrumented compiled Java classes. The model is checked using SPIN to determine if the current execution is on a path that can lead to a requirement failure. ReqMon has been applied to distributed web services. ReqMon automates the translation of monitor specifications into a monitor implementation. Templates for monitor specifications are specialized through the selection of appropriate parameters. At runtime, the monitors track web service traffic at the transport protocol, routers, and gateways. Integrative monitors combine information from individual monitors, and alerts are forwarded to specified websites.

5.18 Sentry System

The Sentry System [11] is a monitoring approach designed for sequential and concurrent C programs. The "sentry" is an observer that is implemented as a separate process to concurrently monitor the execution of a target program and issue a warning if it does not behave correctly with respect to a given set of properties. The properties are specified in a propositional calculus that is extended to include integer-arithmetic operators and relations using C syntax. Universal and existential quantifiers are permitted, as well as summation over a fully instantiated subrange. Annotations are written within specially formatted comments in the places where the properties should be evaluated. The original source file with the annotations is replaced by calls to macros. Properties observed by the global sentry are invariants that are maintained throughout the entire program execution. The local sentry checks properties at a particular point in the execution process.

The local sentry runs in parallel with the source program. The source program sends its variables to the sentry as it executes. The global sentry continuously observes and evaluates the global properties of the source program, waiting until a process writes new state information. The sentry then reads new variables from their buffers, preserving mutual exclusion with the processes, and

enables the checking of any property involving those variables. After the sentry has read the available state information from all processes, it checks the enabled properties. Detected violations are reported by sending a signal to the program. The program initiates a user-defined recovery action based on the type of fault.

5.19 Temporal Rover

Temporal Rover [19], [21] is a specification-based verification tool that uses Linear-Time Temporal Logic (LTL) and Metric Temporal Logic (MTL), allowing the user to specify future time temporal formulae as well as lower and upper bounds, and ranges for relative-time and real-time properties. In Temporal Rover, the user determines the point at which a property should be checked and inserts an annotation of the property. The Temporal Rover parser converts an annotated program into an identical program with the properties implemented in source code. During application execution, the generated code validates the executing program against the formal specifications. Temporal Rover takes a Java, C, C++, Verilog, or VHDL source code program as input. It enables customizable actions for the program domain.

Temporal Rover is a general-purpose monitoring approach, but can support category-specific programs as it has a special code generator targeted for embedded applications and concurrent systems. The user has the flexibility of choosing whether to use a single process approach or generate "target/host" code, where the verification code is generated to be executed in a separate process or machine [21].

6 Related Work

There are numerous tools that have goals or characteristics that are closely related to runtime monitors, but were not identified as having all of the essential elements needed to qualify as a runtime monitor as described earlier. A summary of these tools is presented here.

Dynascope [71] is a programming environment that enables a program, called a director, to monitor and control another program that is being executed by a process called an executor. An ANSI C program is translated to code for the host processor and to hypothetical code, i.e., code compiled for a hypothetical RISC-type processor. An interpreter translates the hypothetical code and provides the framework for directing. The director receives a stream of events from the executor that describe the execution. The director can control the executor by filtering the data, changing its data values, and analyzing computational results. The monitoring afforded by this approach includes such things as capturing the dynamic sequence of function calls, performing periodic analysis of simulations, and debugging.

Flow Analysis for Verification of Systems (FLAVERS) [14] is a finite-state verification, static analysis approach based on using efficient data flow analysis techniques to determine if all possible executions of a software system adhere to properties specified as sequences of events. The properties are represented as finite state automatons. The prototype does not require enumeration of the entire state space of the system. It is based on user-specified events in source code. If a violation is found, FLAVERS can create traces through the model that cause a violation of the property. FLAVERS supports the verification of concurrent systems and was developed for the analysis of systems written in Ada or Java.

Monitor Generator (MG) [55], [56], [57], [58] is a prototype tool that generates modules of a monitor from a Software Requirements Document. MG extends the Software Cost Reduction (SCR) approach that describes system behavior through specifications of finite state automata. The behavior of a system is described by a set of mode classes that are concurrently executing finite state automata in which each state corresponds to a systems mode. Transitions in the finite state automata are triggered by events. The value of each controlled variable is specified by a function defined in terms of the system modes, monitored variables, and terms. Monitoring is restricted to safety properties.

The SPIN [37] model checker is a verification system that supports the design and verification of finite state asynchronous and synchronous process systems. This dynamic analysis approach accepts correctness properties expressed in linear temporal logic (LTL). The verification process either proves that certain behaviors are impossible or it provides examples of behaviors that match. Another model checking program, Java PathFinder (JPF2) [6], [32], [33] model checks Java byte code. It uses static analysis techniques to guide the model checker's search. It also uses runtime analysis to detect possible race conditions. This analysis can be used to guide the model checker. For example, the tool can be run in simulation mode examining a single trace. If a deadlock or data race potential is detected along this single trace, then the model checker focuses on those threads that are involved in the deadlock or data race warning.

The integration of the dynamic invariant detector, Daikon, with the static verifier, ESC/Java (Extended Static Checking) [54] is an approach that combines the strengths of static and dynamic analysis. Daikon is used to detect invariants by running the target program over its test suite, which are then inserted into the target program as annotations. ESC/Java is used next to check the annotated target program and to determine which of the invariants can be statically verified. ESC/Java checks portions of the target program independently. It can statically detect common errors that are usually not detected until runtime, e.g., null dereference, array bounds, and type cast errors. Another static and dynamic approach is called Enforcing Trace Properties by Program Transformation [15]. In this approach, the programmer specifies properties on programs separately and the program transformer takes the program and the properties and produces another program satisfying the property. The system then tries to detect statically if the source program enforces the specified properties by static analysis (e.g., control-flow analysis). Dynamic checks are also used.

Dynamic Assembly from Models (Dynamo) [65], [66] is an environment designed for the assembly of dependable and efficient systems from components. The project

includes a graphical design environment, static analyzers of assembly properties, integrated tools architecture, and a runtime monitoring infrastructure. Dynamic event analysis is used to monitor and analyze properties that cannot be verified statically. System status and resource consumption are examples of runtime information that is provided from the runtime environment.

The Observer [18] is an approach to build systems whose online behavior is checked against a formal model derived from a formal specification. An observer-worker system is a potentially runtime checked system that is constituted of two distinct components: a worker and an observer. The worker implements system behavior. The observer is a redundant implementation whose outputs are compared with the outputs of the worker. The observer models are defined by Petri nets, animated by simulators, and derived as observational models from protocol descriptions. The observer tests for a system that is fault-secure.

Dynamic Assertions Using TXP [22] is designed to support dynamic monitoring of temporal properties at runtime to verify hardware system properties. Assertions are written in a temporal logic and checked statically or dynamically. When a failure of an assertion is detected, the event-handler provides the starting time of the assertion and the time when the failure is detected.

Test and Measurement Processor (TMP) [31] is a hybrid monitor consisting of hardware and software components that capture data about hardware in a multicomputer system. The types of data captured include machine metrics, such as CPU time, idle time, and accumulated ready time, and process metrics, such as elapsed time and time in ready queue. Raw data is processed, analyzed, and evaluated synchronously with the program execution. The triggering points for events are placed in the operating system kernel.

Assertion Definition Language Translator (ADLT) [17] is a system that assists in the functional testing of software components. It is a compiler that generates test programs based upon formal functional and test-data specifications. An auxiliary function is any function which is used in an ADL specification to describe the workings of a function under test, but which is implemented separately from the module being specified. ADL assertions can include input parameters, expected return values, and significant conditions that should be true after a function has completed.

7 Summary

There is renewed interest in the field of dynamic software fault-monitoring largely because systems are becoming more difficult to verify due to increases in size and complexity. The motivation for the survey is twofold: to create a taxonomy that developers and researchers can employ to differentiate between dynamic software fault-monitoring approaches that reflect advancements in the field and to promote the use of monitors now that increases in computing power have made it feasible for systems to support such technology. The work provides a common language for discussing runtime monitoring systems and their components.

More than 30 software fault-monitoring systems were surveyed in the literature. After the systems were classified, the authors of those systems were asked to review and validate each of their respective system's classification within the taxonomy and to provide comments on both the classification of their system and the taxonomy in general. The taxonomy and classifications have been revised based on initial feedback and further analysis.

The classification shows that many of the systems use logic or a high-level to very-high-level language to specify software properties. Systems that use automata and algebras to specify properties are limited. Specifications based on automata are helpful for capturing behavior of event-driven systems. In addition, the use of automata to specify properties lends itself to visualization of the properties that can make the specifications more understandable for customers with limited training in formal methods. An example of such a tool is Propel [69] that uses automata to elucidate specifications.

All surveyed approaches are able to specify and monitor implementation-level properties, while fewer can capture domain-level and design-level properties. With the growth and complexity of software systems, the ability to capture properties prior to implementation of the system is becoming more critical. This is especially true on team projects in which different teams work on analysis, design, and implementation. Domain-level and design-level properties can provide assurance that properties are being met in implementation as well as serve as a communication vehicle among team members.

Of the approaches surveyed, most are applicable to any type of application. There are a few that specifically target distributed and parallel systems, e.g., BEE++, Falcon, JRTM, Sentry, and the noninterference monitoring architecture approach. MaC and DB Rover are the only approaches of those surveyed that use dynamic analysis to monitor properties. There is a large class of properties that requires analysis of multiple state changes or events to determine whether a property is met. The work in this area is important if these properties are to be monitored.

Only the noninterference monitoring architecture approach [75], [76] uses hardware for monitoring. The DynaMICs approach has proposed a snoopy coprocessor approach [73] to monitor critical properties. The argument for shifting critical monitoring to hardware is to prevent errors before they can cause a change in state and to reduce the impact of monitoring on performance. Software monitors detect errors once a change in state has occurred; however, in systems in which a change in state can affect external devices, monitoring at the bus level can detect a change in state before memory is revised. This can prevent a catastrophic error before it occurs. It is natural to move well-established technology into hardware when appropriate. For example, error-correcting codes used to be monitored in software; now it is being done in hardware.

A dream of engineering is to be able to build a system and have confidence that it will behave as intended. In most engineering fields, this dream has been largely realized by incorporating safety features in the design. In software, however, it remains elusive. Unlike other engineering

disciplines, software development often requires building systems that lie outside the realm of the developer's experience. Additionally, the necessity to integrate knowledge from multiple domains increases the complexity of many systems being developed today. Although significant gains have been made in the production of high-quality software, there is no guarantee that software will execute without failures. The goals of software-fault monitoring are similar to the goals of other assurance techniques, namely, to provide evidence of the correctness of a given program execution. Such an approach is useful in the detection and isolation of faults and in reducing the gap between faults and failures. This paper summarizes the research in the area of software fault monitoring and presents some recent developments in the area that move the state-of-the-art toward realization of the dream.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their insightful remarks and suggestions. They would also like to acknowledge the researchers who replied to their request to review and comment on the classification of their tools. This research was partially supported by NASA project no. NCC5-498 and US National Science Foundation project nos. EAR-0112968, EAR-0225670, and EIA-0321328.

REFERENCES

[1] B. Alpern and F. Schneider, "Verifying Temporal Properties without Temporal Logic," *ACM Trans. Programming Languages,* vol. 11, no. 1, pp. 147-167, Jan. 1989.
[2] A. Avizienis and J. Laprie, "Dependable Computing: From Concepts to Design Diversity," *Proc. IEEE,* vol. 74, no. 5, pp. 629-638, May 1986.
[3] M. Barbacci, "Quality Attributes," Technical Report CMU/SEI-95-TR-021, Software Eng. Inst., Carnegie Mellon Univ., Dec. 1995.
[4] D. Bartetzko, "Jass—Java with Assertions," *Proc. First Workshop Runtime Verification (RV '01),* July 2001.
[5] B. Berard and M. Bidoit, *Systems and Software Verification: Model-Checking Techniques and Tools.* Springer Verlag, 2001.
[6] G. Brand, "Java Pathfinder JPF2 Second Generation of Java Model Checker," http://www.ist.tugraz.at/courses/akswt03/paper_jpf2.pdf, July 2004.
[7] B. Bruegge, T. Gottschalk, and B. Luo, "A Framework for Dynamic Program Analyzers," *Proc. Eighth Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 65-82, 1993.
[8] L. Burdy, Y. Cheon, D.R. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K. Rustan, M. Leino, and E. Poll, "An Overview of JML Tools and Applications," *Proc. Eighth Int'l Workshop Formal Methods for Industrial Critical Systems (FMICS '03),* T. Arts and W. Fokkink, eds., June 2003.
[9] F. Chen and G. Roşu, "Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation," *Electronic Notes in Theoretical Computer Science,* vol. 89, no. 2, 2003.
[10] Y. Cheon, "A Runtime Assertion Checker for the Java Modeling Language," Technical Report 03-09, Dept. of Computer Science, Iowa State Univ., Ames, 2003.
[11] S. Chodrow and M. Gouda, "Implementation of the Sentry System," *Software Practice and Experience,* vol. 25, no. 4, pp. 73-387, 1995.
[12] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, "Principles of Maude," *Electronic Notes in Theoretical Computer Science,* vol. 4, 1996.
[13] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesada, "Using Maude," *Proc. Third Int'l Conf. Fundamental Approaches to Software Eng.,* 2000.
[14] J. Cobleigh, L. Clarke, and L. Osterweil, "Flavers: A Finite State Verification Technique for Software Systems," technical report, Univ. of Massachusetts Amherst, Apr. 2001.
[15] T. Colcombet and P. Fradet, "Enforcing Trace Properties by Program Transformation," *Proc. 27th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 54-66, 2000.
[16] N. Delagdo, "A Taxonomy of Dynamic Software-Fault Monitoring Tools," master's thesis, Univ. of Texas at El Paso, May 2002.
[17] J. deRaeve and S. McCarron, "Automated Test Generation Technology: Assertion Definition Language Project," technical report, X/Open Company Ltd., 1997.
[18] M. Diaz, G. Juanole, and J. Courtiat, "Observer—A Concept for Formal On-Line Validation of Distributed Systems," *IEEE Trans. Software Eng.,* vol. 20, no. 12, pp. 900-913, Dec. 1994.
[19] D. Drusinsky, "The Temporal Rover and the ATG Rover," *Proc. SPIN 2000 Conf.,* 2000.
[20] D. Drusinksky, "Temporal Rule Checking," An Oracle Apps-World Online White Paper, http://www.dbrover.com, July 2004.
[21] D. Drusinsky, Time-Rover, Cupertino, Calif., personal comm., 2002.
[22] S. Dudani, J. Geada, G. Jakacki, and D. Vainer, "Dynamic Assertions Using TXP," *Proc. First Workshop Runtime Verification (RV '01),* July 2001.
[23] G. Eisenhauer, "The Echo Event Delivery System," http://www.cc.gatech.edu/systems/projects/ECho, July 2004.
[24] G. Eisenhauer, Georgia Inst. of Technology, Atlanta, Ga., personal comm., 2002.
[25] G. Eisenhauer and K. Schwan, "The Mirror Object Steering System," http://www.cc.gatech.edu/systems/projects/MOSS, July 2004.
[26] A. Gates and O. Mondragon, "A Constraint-Based Tracing Approach," *J. Systems and Software,* vol. 63, pp. 219-239, Sept. 2002.
[27] A. Gates, S. Roach, O. Mondragon, and N. Delgado, "DynaMICs: Comprehensive Support for Run-Time Monitoring," *Proc. First Workshop Runtime Verification (RV '01),* July 2001.
[28] A. Gates and P. Teller, "An Integrated Design of a Dynamic Software-Fault Monitoring System," *J. Integrated Design and Process Science,* vol. 14, pp. 63-78, 2000.
[29] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, and J. Vetter, "Falcon: On-Line Monitoring and Steering of Large-Scale Parallel Programs," Technical Report GIT-CC-94-21, College of Computing, Georgia Inst. of Technology, Apr. 1994.
[30] W. Gu, J. Vetter, and K. Schwan, "An Annotated Bibliography of Interactive Program Steering," *ACM SIGPLAN Notices,* Sept. 1994.
[31] D. Haban and D. Wybranietz, "A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems," *IEEE Trans. Software Eng.,* vol. 16, no. 2, pp. 197-211, Feb. 1990.
[32] K. Havelund, "Using Runtime Analysis to Guide Model Checking of Java Programs," *Lecture Notes in Computer Science,* vol. 1885, pp. 245-264, 2000.
[33] K. Havelund, NASA, Ames, Moffet Field, Calif., personal comm., 2002.
[34] K. Havelund and G. Roşu, "Monitoring Programs Using Rewriting," *Proc. 16th IEEE Int'l Conf. Automated Software Eng.,* pp. 135-143, 2001.
[35] K. Havelund and G. Roşu, "Java PathExplorer—A Runtime Verification Tool," *Proc. Symp. Artificial Intelligence, Robotics and Automation in Space,* June 2001.
[36] K. Havelund and G. Roşu, "Monitoring Java Programs with Java PathExplorer," *Proc. First Workshop Runtime Verification (RV '01),* July 2001.
[37] G. Holtzmann, "The Spin Model Checker," *IEEE Trans. Software Eng.,* vol. 23, no. 5, pp. 279-295, May 1997.
[38] C.L. Jeffery, *Program Monitoring and Visualization: An Exploratory Approach.* Springer-Verlag, 1999.
[39] C.L. Jeffrey, New Mexico State Univ., Las Cruces, N.M., personal comm., 2002.
[40] C.L. Jeffery, W. Zhou, K. Templer, and M. Brazell, "A Lightweight Architecture for Program Execution Monitoring," *Proc. ACM SIGPLAN/SIGSOFT Workshop Program Analysis for Software Tools and Eng.,* pp. 67-74, 1998.
[41] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Java-MaC: A Run-Time Assurance Tool for Java Programs," *Proc. Fourth IEEE Int'l High Assurance Systems Eng. Symp.,* pp. 115-132, 1999.
[42] M. Kim, I. Lee, and O. Sokolsky, Univ. of Pennsylvania, Philadelphia, personal comm., 2002.
[43] M. Kim and M. Viswanathan, "MaC: A Framework for Run-Time Correctness Assurance of Real-Time Systems," Technical Report MS-CIS-98-37, Dept. of Computer and Information Sciences, Univ. of Pennsylvania, Dec. 1998.

[44] M. Kim and M. Viswanathan, "Formally Specified Monitoring of Temporal Properties," *Proc. European Conf. Real-Time Systems,* 1999.
[45] I. Lee and H. Ben-Abdallah, "A Monitoring and Checking Framework for Run-Time Correctness Assurance," *Proc. 1998 Korea-U.S. Technical Conf. Strategic Technologies,* 1998.
[46] I. Lee and M. Kim, "Runtime Assurance Based on Formal Specifications," *Proc. 1999 Int'l Conf. Parallel and Distributed Processing Techniques and Applications,* 1999.
[47] Y. Liao and D. Cohen, "A Specificational Approach to High Level Program Monitoring and Measuring," *IEEE Trans. Software Eng.,* vol. 18, no. 11, pp. 969-978, Nov. 1992.
[48] D. Luckham, W. Mann, S. Meldal, and D. Helmbold, "An Overview of Ann Specification Language for Ada," *IEEE Software,* vol. 20, pp. 9-23, 1988.
[49] D. Luckham, S. Sankar, and S. Takahashi, "Two-Dimensional Pinpointing: Debugging with Formal Specifications," *IEEE Software,* vol. 8, no. 1, pp. 74-84, Jan. 1991.
[50] B. Meyer, "Applying Design by Contract," *Computer,* vol. 25, no. 10, pp. 40-51, Oct. 1992.
[51] A. Mok, Univ. of Texas at Austin, Austin, Tex., personal comm., 2002.
[52] A. Mok and G. Liu, "Efficient Run-Time Monitoring of Timing Constraints," *Proc. Third IEEE Real-Time Technology and Applications Symp.,* pp. 252-262, 1997.
[53] M. Möller, Univ. of Oldenburg, Oldenburg, Germany, personal comm., 2002.
[54] J. Nimmer and M. Ernst, "Static Verification of Dynamically Detected Program Invariants: Integrating Daikon and ESC/Java," *Proc. First Workshop Runtime Verification (RV '01),* July 2001.
[55] D. Peters, "Automated Testing of Real-Time Systems," technical report, Memorial Univ. of Newfoundland, Nov. 1999.
[56] D. Peters, "Deriving Real-Time Monitors from System Requirements Documentation," PhD thesis, McMaster Univ., Jan. 2000.
[57] D. Peters, Memorial Univ. of Newfoundland, St. John's, Canada, personal comm., 2002.
[58] D. Peters and D. Parnas, "Requirements-Based Monitors for Real-Time Systems," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA),* pp. 77-85, 2000.
[59] B. Plattner and J. Nievergelt, "Monitoring Program Execution: A Survey," *Computer,* vol. 14, pp. 76-93, 1981.
[60] W.N. Robinson, "Monitoring Web Service Requirements," *Proc. 12th IEEE Int'l Conf. on Requirements Eng.,* pp. 65-74, 2003.
[61] W.N. Robinson, "Monitoring Software Requirements Using Instrumented Code," *Proc. Hawaii Int'l Conf. System Sciences,* vol. 9, pp. 276-287, Jan. 2002.
[62] D. Rosenblum, "A Practical Approach to Programming with Assertions," *IEEE Trans. on Software Eng.,* vol. 21, no. 1, pp. 19-31, Jan. 1995.
[63] G. Roşu and K. Havelund, "Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae," RIACS Technical Report TR 01-15, May 2001.
[64] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing," Health, Social, and Economic Research, Project No. 7007-001, Research Triangle Park, N.C., 2002.
[65] S. Rugaber, "DYNAMO Dynamic Assembly from Models," http://www.cc.gatech.edu/dynamo/txt/proposal.ps, July 2004.
[66] S. Rugaber and R. Stirewalt, "Position Paper: The Tradeoff between Dependability and Efficiency in Embedded Systems," *Proc. CASES 2001 Conf.,* Aug. 2001.
[67] S. Sankar and M. Mandal, "Concurrent Runtime Monitoring of Formally Specified Programs," *Computer,* vol. 26, no. 3, pp. 32-41, Mar. 1993.
[68] B. Schroeder, "On-Line Monitoring: A Tutorial," *Computer,* vol. 28, no. 6, pp. 72-78, June 1995.
[69] R. Smith and G. Avrunin, L. Clarke, and L. Osterweil, "PROPEL: An Approach Supporting Property Elucidation," *Proc. Int'l Conf. Software Eng.,* May 2002.
[70] O. Sokolsky, S. Kannan, and M. Viswanathan, "Steering of Real-Time Systems Based on Monitoring and Checking," *Proc. Fifth Int'l Workshop Object-Oriented Real-Time Dependable Systems,* pp. 11-18, 1999.
[71] R. Sosic, "Dynascope: A Tool for Program Directing," *Proc. Fifth ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 12-21, 1992.
[72] A. Tanenbaum, *Modern Operating Systems.* Prentice Hall, 2001.
[73] P. Teller, M. Maxwell, and A. Gates, "Towards the Design of a Snoopy Coprocessor for Dynamic Software-Fault Detection," *Proc. 18th IEEE Int'l Performance, Computing, and Comm. Conf.,* pp. 310-317, 1999.
[74] K. Templer and C. Jeffery, "A Configurable Automatic Instrumentation Tool for ANSI C," *Proc. 13th IEEE Int'l Conf. Automated Software Eng.,* pp. 249-258, 1998.
[75] J. Tsai, Y. Bi, S. Yang, and R. Smith, *Distributed Real-Time Systems: Monitoring, Visualization, Debugging and Analysis.* John Wiley and Sons, 1996.
[76] J. Tsai and K. Fang, "A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging," *IEEE Trans. Software Eng.,* vol. 16, no. 8, pp. 897-916, Aug. 1990.

Nelly Delgado received the BS and MS degrees in computer science from the College of Engineering at the University of Texas at El Paso (UTEP). She worked as a lecturer at UTEP from 2002-2003. She is currently employed as a consultant with Wadeware LLC in Bellevue, Washington. She serves as secretary for the IEEE-CS Educational Activities Board. She is a student member of the IEEE and the IEEE Computer Society.

Ann Quiroz Gates received the BS degree in mathematics from the College of Science and the MS degree in computer science from the College of Engineering at the University of Texas at El Paso (UTEP). She received the PhD degree in computer science from the College of Arts and Sciences at New Mexico State University. She is currently employed as an associate professor at UTEP. Her research interests are in software-fault monitoring and grid computing. She is a member of the US National Science Foundation funded GEON research team that is building the cyberinfrastructure for the geosciences. She is a member of the IEEE Computer Society, an IEEE Computer Society Certified Software Development Professional, and a member of the IEEE Computer Society Board of Governors.

Steve Roach received the BS degree in Chemistry from Ohio University, and the MS and PhD degrees from the University of Wyoming. He has been a member of the Automated Software Engineering Group at NASA Ames, where he worked on the deductive synthesis system, Amphion. He is currently employed as an assistant professor at the University of Texas at El Paso. He has experience in automated theorem proving, synthesis, and integration of decision procedures in theorem provers. He has worked for a variety of chemical engineering firms developing data acquisition and process control systems and process modeling software. He is a principal author of a scientific opportunity visualization program for the Cassini mission to Saturn. He is a member of the IEEE Computer Society and is an IEEE Computer Society Certified Software Development Professional.