DynaMICs: Comprehensive Support for Run-Time Monitoring

A. Q. Gates, S. Roach, O. Mondragon, and N. Delgado [1,2]

Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, USA

Abstract

Software engineering strives to enable the economic construction of software systems that behave reliably, predictably, and safely. In other engineering disciplines, safety is assured in part by detailed monitoring of processes. In software, we may achieve some level of confidence in the operation of programs by monitoring their execution. DynaMICs is a software tool that facilitates the collection and use of constraints for software systems. In addition, it supports traceability by mapping constraints to system artifacts. Constraint specifications are stored separately from code; constraint-monitoring code is automatically generated from the specifications and inserted into the program at appropriate places; and constraints are verified at execution time. These constraint checks are triggered by changes made to variable values. We describe the architecture of DynaMICs, discuss alternative verification techniques, and outline research directions for the DynaMICs project.

1 Introduction

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. 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.

Significant gains have been made in the production of high-quality software through the application of process guidelines such as the Capability Maturity

[1] This work is supported by NASA grants NAG2-1012, NAG2-1138, and NCC5-205, Sandia grant BD-9421, CONACYT-68761, and NSF grants EIA-9522207 and EIA-9729990. The authors would like to thank the reviewers for their constructive remarks.

[2] Email:agates, sroach, oscar, ndelgado@cs.utep.edu

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Model (CMM), Personal Software Process (PSP), and International Standardization Organization (ISO) 9000. These guidelines focus on the process of software construction. There are no assurances that use of these approaches will yield quality software. For instance, there is no guarantee that software produced by an organization at CMM level 5 will execute without failures. Many approaches to software verification exist (some are described in Section 3); however, short of exhaustive testing, none can show that execution of code for a complex system is correct for all possible inputs.

Chemical, industrial, electrical, and civil engineers build reliable and safe systems by incorporating safety features in their designs. For example, boilers are built with pressure-relief valves, distillation columns have high-temperature shutoffs, and bridges are constructed so that the predicted failure load is several times the predicted maximum load. In situations where construction alone does not ensure safe use, such as in a chemical manufacturing operation, the system is monitored during use. The software analogue to this is run-time monitoring. While assertions and exception handling have been in use for over twenty-five years, more general run-time monitoring has not been widely adopted as standard practice mainly because of performance degradation and the difficulty in maintaining programs that contain checking code.

The goals of run-time monitoring are similar to the goals of other assurance techniques, namely to provide evidence of the correctness of a given program execution. In run-time monitoring, however, we are not trying to prove that the program will execute correctly in every situation or for all inputs. We are only trying to demonstrate that the current execution of the program is correct with respect to a set of specified properties. This simplifies the problem to some extent. We are not concerned about all of the possible execution paths or states that a program could possibly visit. We are only concerned with those states that the program actually does visit in a given execution. In each of these states, we attempt to demonstrate that the critical properties of the program are maintained. Such an approach should prove useful in the detection and isolation of faults, and in reducing the gap between faults and failures. By monitoring critical program properties at runtime, faults should be detected before they lead to serious failures. Another use is to improve the probability of detecting faults during testing.

The standard approach to run-time monitoring is a time-consuming process. Typically, a program is analyzed with respect to a specification, frequently in the context of attempting to isolate an error. Constraints and variables that indicate key elements of state are identified. Code to test constraints or monitor the variables is inserted, and then the instrumented program is executed. The outputs are analyzed, and either a correction is made, or the process is repeated. The difficulties with this approach are that it is quite time consuming, the instrumentation of the code may make maintenance more difficult, and generally, the constraints and events that are monitored are chosen ad-hoc in order to isolate an error.

Phases	Extended Process	Driving Inquiries	Distinguishing Features
Requirements Analysis and Specification	Elicit and specify constraints from domain experts and users	What behavior can be monitored to detect correct execution?	* Captures constraints and maintains them separate from the program * Automates insertion of constraints * Provides support for constraint maintenance * Determines impact of change by monitoring constraints on the modified code
Development	Capture assumptions and limitations from developers	What assumptions are being made about the use of the software?	
Testing	Identify test-specific constraints	What state information needs to be monitored during testing?	
Maintenance	Create, update, or delete constraints	What constraints have changed or need to be added?	

Table 1
Extensions and features provided by DynaMICs.

In an approach called DynaMICs [13,17,18,19], we are addressing these issues by providing tools that support the construction and insertion of constraint monitoring code and facilitate tracing of failures to requirements. The key features of DynaMICs are that constraints are elicited at each of the requirements, design, and implementation levels; constraints are stored separately from code; constraint-testing code is automatically inserted into programs at the object-code level; and a tracing mechanism provides relevant information about the violated constraints. A distinguishing characteristic of our constraint definitions is that constraints need not be implementation-aware. DynaMICs provides tools to assist in the specification of constraints and the linking required for tracing. Table 1 summarizes the extensions to the software process that are required by the DynaMICs system, gives the inquiries that drive constraint elicitation, and summarizes the features of this approach.

This work is significant because the approach focuses on identifying constraints that assure correct behavior at runtime and using these to detect errors that may be difficult to detect using other approaches. Absence of detected constraint violations indicates preservation of constraints (at least at the points in code where the constraint is tested). The use of tracing facilitates the correction of errors by maintaining the relations among code, constraints, and supporting documentation. Although the approach is less complete than model checking or theorem proving, it complements testing by showing that the program is working correctly for the current execution with respect to the monitored properties.

The next section of the paper describes the approach in more detail. Section 3 briefly discusses other approaches to verification. Section 4 discusses future directions in the project. The paper ends with a summary in Section 5.

2 DynaMICs Approach

DynaMICs distinguishes itself from other approaches by keeping constraints separate from the code, automating the generation of constraint-checking code, automating the instrumentation of the program with constraint-checking code or directives, and providing support for tracing between constraints, code, and documentation. Figure 1 presents a dataflow diagram of the system, where the arcs denote flow of data, ovals denote data transformation, shaded boxes denote sources or sinks, and open-ended rectangles denote data stores. The shading on the ellipses distinguishes between those transformations that are done statically, i.e., prior or subsequent to program code execution and those that are done dynamically, i.e., during program code execution. Definition of constraints and knowledge, initialization of the monitor, linking constraints with its supporting documentation, and static analysis of constraints are done before execution. Initialization of the monitor includes control-flow analysis of the program code, checking-code generation, and instrumentation. Monitoring and tracing are done dynamically. Tracing can also be done after program execution through use of state history generated at run time. Each of the major transformations, i.e., definition of constraints and knowledge, initialization of the monitor, monitoring, and tracing and analysis, is discussed in the subsections below.

2.1 Constraint and Knowledge Definition

Constraints capture domain knowledge such as properties, behaviors, and relationships among real-world objects being modeled by the software as well as assumptions and limitations of the implementation. Unlike requirements elicitation that asks the question, "What will the system do?" constraint elicitation asks the question "What properties of modeled objects indicate correct program execution?" For example, consider the division of two integers, a and b, that yields a quotient, q, and a remainder, r. For this problem, two constraints can be defined: $r < b$ and $(q \times b) + r = a$. The constraints do not recalculate the division of a and b, rather they check that the division is correct. Typically, constraints are gleaned from requirements. In addition, analysts can identify constraints through interviews with domain experts using approaches such as Fault Tree Analysis [29] to direct the questions. Other sources of constraints include users, customers, and developers.

As shown in Figure 1, constraint specifications are stored in a *constraint and knowledge repository*. Because constraints can be defined independently

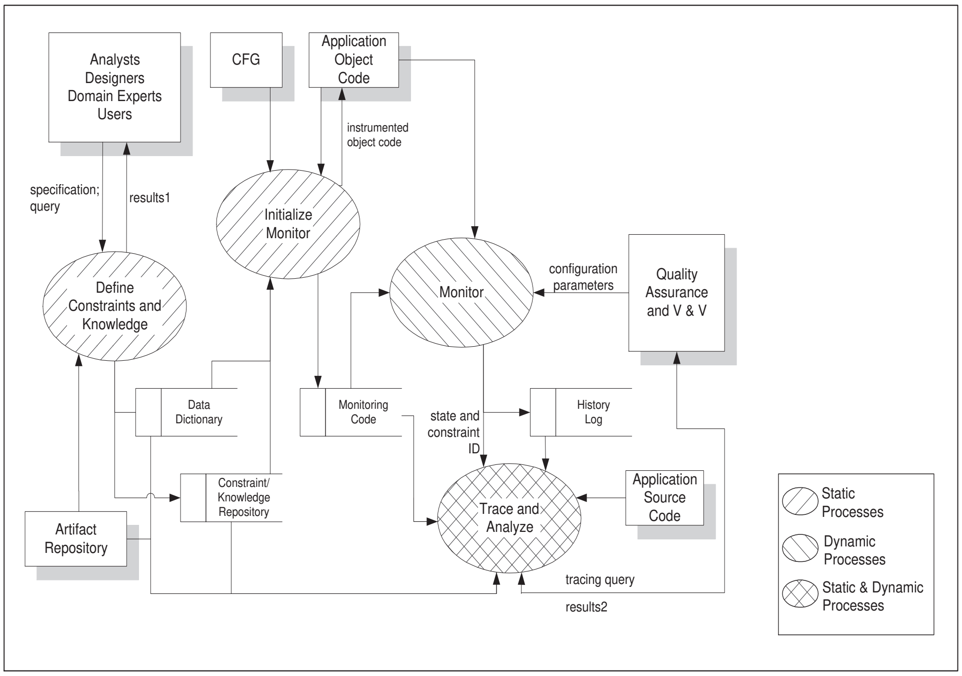

Fig. 1. Data flow in a DynaMICs system.

of the implementation, it is necessary to maintain a *data dictionary* that stores the types and descriptions of the variables used in the specifications. Constraint variables are mapped manually to code variables, and this mapping is saved in the data dictionary. It is desirable to automate the translation between variables used in constraints to those used in code.

As constraints are specified, they are mapped to supporting documentation, such as interview transcripts, SRS documents, software design documents, memoranda, or correspondences that are stored in an *artifact repository*. Other pertinent information, such as author and date, is maintained with the constraint specification.

Constraints are specified as event-condition-action rules [19]. The event directs automated program instrumentation. The condition and action are used to generate constraint-checking code. When additional information is needed to compute the value of a constraint, this information must also be specified. As with a constraint specification, the event directs automated program instrumentation. When the condition holds, the action results in a store to a special variable that has been declared in the data dictionary. For example, in a program that parses an input string, it may be necessary to count the occurrences of the '#' character to check a constraint that places an upper bound on the number of '#' characters that may be contained in the string. A user would need to declare a special variable that maintains the count and specify an event whose action is to increment this variable when the '#' character is read. A brief description of events, conditions, and actions follows.

Event. The event definition is defined as an ordered four-tuple [16,17]:

Variable-set x Transition x Phase x Placement, where

Variable-set:	set-of-tokens
Transition:	{Immediate, Intermediate, Delayed}
Phase:	{Input, Processing, Output}
Placement:	{Before-store, After-store}

In DynaMICs, we define *state* to be the collection of values held in constraint variables. A state transition is a change in some constraint variable. The definition of an event includes the definition of what state transitions need to be monitored and when those transitions need to be monitored. The event definition restricts constraint checks to certain execution paths. The *Variable-set* of the event definition identifies a set of constraint-variables to be monitored, which must be mapped to implementation-level variables. Decorated variables are used to differentiate initial, previous, and current state.

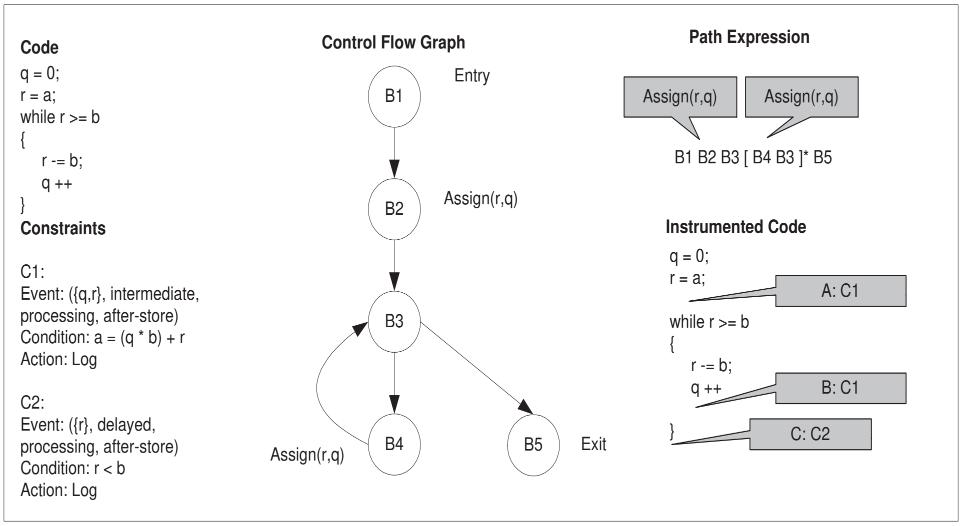

Fig. 2. Constraint instrumentation for the division example.

Transition, Phase, and *Placement* describe when a constraint check should occur. *Phase* is used to restrict the testing of constraints. For example, when *Phase* is *Input*, the constraint is tested when a variable is updated by an input operation. If the *Phase* is *Processing*, the constraint is tested when an assignment is made to a variable. If the *Phase* is *Output*, the constraint is tested when a variable is used in an output system call. Because computed values may not be stored to memory, but instead stored to a device via output instructions, it also is necessary to consider output instructions as instructions that cause state transitions. For a specified *Phase* and *Variable-set*, an *Immediate* transition indicates that the constraint must hold after each state transition. *Delayed* denotes that a constraint must hold at the end of a specified phase. For example, a constraint for variables *a, b*, and *c* that is specified as *Delayed* and *Input* for *Transition* and *Phase*, respectively, indicates that the associated monitoring code executes after all values for these variables have been read. If a program uses an iterative construct to read in these values, then the check will occur at the point where the iterative construct terminates. In the case of nested iterative constructs, the check will occur outside the outermost construct. An *Intermediate* transition designates that the constraint must hold after some sequence of state transitions. Figure 2 provides an example using the constraints identified for the division example given in Section 2.1. In the figure, note that constraint *C1* (*Intermediate* and *Processing*) is checked each time a value is stored to r and q (see labels *A:C1* and *B:C1* in Figure 2). Constraint *C2* is checked after all stores to r, i.e., when the *while* loop terminates (see label *C:C2* in Figure 2).

We assume that controls of sensitive devices are memory mapped and, thus, are accessed using assignments [19]. *Placement* indicates whether a constraint is placed before a store or after a store to a constraint variable. In the case of a constraint in which a violation will cause catastrophic failure (referred to as a *mission-critical* constraint), it is imperative that the constraint is checked before the value is stored. For constraints that are not mission-critical, the test may occur after a write to a variable and, thus, can be checked in parallel to reduce the impact on performance.

Condition. A condition definition is expressed in a first-order language [12]. In addition to specifying relationships between program variables, conditions can determine whether the following hold: the value associated with a program variable is within a range of values or is a member of a set of values, a program variable has been assigned a value prior to use, two sets of values are disjoint (e.g., no part-time employee is classified as a full-time employee), or a set is a subset of another (e.g., all scholarship recipients are registered as full-time students).

Action. The action definition specifies the consequence of a constraint violation. This can include such actions as recording state in a history log, saving state for error recovery, performing state rollback, or initiating graceful degradation. The latter three actions are currently under study. See Figure 3 for an example constraint. In this constraint, variable *Nkpower* is being monitored. Placement value *Before-store* indicates that the constraint is checked immediately prior to a value being stored in *Nkpower*. The constraint, which is given in the condition, states that *Stable(X)* must hold if *On(Nkpower)* holds in the intercepted update.

2.2 Monitor Initialization and Execution

Program code can be instrumented with constraint-checking code using an in-line sequence of instructions, a function call, or a trigger that initiates the constraint check on a separate process [9]. As described earlier, the event definition of a specification along with analysis of a program's execution flow

Constraint:	The system shall ensure that X must be stable prior to commanding on the NK power supply.
Event:	({Nkpower}, Immediate, Processing, Before-store}
Condition:	On(Nkpower) -> Stable(X)
Action:	Log

Fig. 3. Example specification.

directs the points in program execution at which constraint checks should be executed [14].

The analysis approach followed by DynaMICs focuses on *path expressions* [1,25,41], i.e., regular expressions derivable from execution control-flow graphs (CFGs), each node of which is a basic block that has been annotated with read/write lists (see Figure 2). These expressions are analyzed to identify program instrumentation points. Instrumentation points are important because they identify locations in code where variables-of-interest (VOIs) used in constraints are modified. For each constraint there is a set of instrumentation points. A general algorithm for defining instrument points for constraints can be found in [17]. The algorithm does not consider all classes of intermediate constraints. Extensions to the algorithm are under study. Figure 2 illustrates the instrumentation process for the division example given in Section 2.1.

Analysis can be done at the source-code, intermediate-code or object-code level. In memory-mapped I/O systems, it may be necessary to prevent a write to memory prior to testing a constraint. Because safety-critical systems require assurance that is not provided by current compilers, checking of mission-critical constraints requires instrumentation at the object-code level.

The role of the monitor is delegated either to the process executing the application code, or to another process not necessarily on the same processor. When a separate process is used, enough state information to verify the constraint must be communicated, and for mission-critical constraints, synchronization between processes is necessary [42]. The monitor observes the executing program and, when it detects a constraint violation, it executes the action associated with the constraint. The action can be to record state and notify the tracing and analysis component of DynaMICs.

2.3 Tracing and Analysis

Tracing [7] is the activity of following links between artifacts to identify and document the origin and decomposition of artifacts during the development process. For example, a requirement in a requirements specification document may be linked to a method in software source code. Tracing provides the ability to demonstrate completeness, necessity, and consistency of artifacts used during software system development and that an artifact contains or implements all requirements of the predecessor artifact.

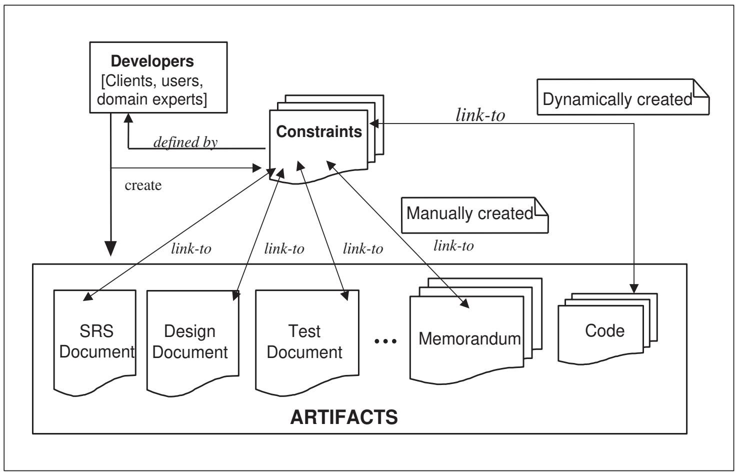

Fig. 4. High-level FasTLInC traceability model.

Unlike traditional traceability models, the tracing approach used in DynaMICs [15], called *constraint-based tracing*, focuses on constraints. Tracing is performed by a component of DynaMICs called Fast Tracing with Links using Integrity Constraints (FasTLInC). The FasTLInC subsystem establishes the relationship between constraints and artifacts and facilitates analysis of violations. The use of constraints to automatically define links to code and to capture run-time information distinguishes constraint-based tracing approach from others. FasTLInC traces by traversing the path from application code to constraints and from constraints to artifacts. Figure 4 presents a high-level view of the constraint-based traceability model. Although FasTLInC does not support tracing among artifacts as does traditional tracing approaches, it can be integrated with existing tools such as TOOR [33] or SODOS [23] to provide this functionality. Because DynaMICs verifies that a program behaves in accordance with constraints, the traceability provided by FasTLInC is significant since the monitor targets the detection of faults that result from ambiguity and changes in requirements, conflicts among requirements, and change in program use. The automated identification of bi-directional links between constraints and code eliminates the laborious task of managing links, which can be problematic because of the evolutionary nature of code. In addition, the approach supports constraint management, change management, error-source identification, and resolution of conflict among constraints and artifacts.

FasTLInC serves as a database management system for constraints, knowledge, and data dictionary entries and allows a user to query this database. Constraints can be grouped together to form *working sets*, i.e., groupings of constraint specifications that share common attributes such as date, author, variable names, or aspects of the event classification. By viewing working sets, the developer can examine constraints to check for inconsistencies among them as well as examine knowledge collected from multiple sources and correct possible discrepancies early in the lifecycle [13].

FasTLInC interacts with both the artifact and history-log repositories and the monitoring code. FasTLInC assists tracing and supports reconciliation of violations by providing access to the data managed by the system. A history log allows the user to trace through and study program behavior when violations are detected, or after the program has completed its execution. In DynaMICs, any constraint and its associated artifacts can be linked to the program-execution point at which the constraint check or knowledge computation occurs. Conversely, FasTLInC provides the capability to search on scope, i.e., to retrieve constraints that check a specified section of code.

3 Related Work

We partition techniques that can be used to provide evidence of correct program execution into two main categories: those that are applied before a software product is deployed and those that are applied after the product is deployed. The pre-deployment techniques include activities that are used prior to the delivery and installation of the software system. The post-deployment techniques are those that are packaged and installed with the software system and are used during execution to evaluate the success and progress of execution. Each of these categories is discussed in the subsections below with a focus on automated techniques.

3.1 Pre-Deployment Techniques

Pre-deployment techniques include those that are applied statically (e.g., program synthesis, correctness proofs, code analysis, and reviews) and those that are applied dynamically (e.g., testing and model checking).

Program synthesis. Approaches such as deductive synthesis and transformational synthesis address correctness by constructing provably correct software [10,32]. Although these systems produce high-quality and high-assurance software, they are costly. Many of the techniques and tools currently used depend on formal specification of the software's behavior and formal specification of domain-specific knowledge. It is, in general, difficult to collect and specify this knowledge. It is possible, and even likely, that errors will be made in the specifications. Finally, if the tool depends on automated reasoning involving search, the search space tends to be exponential. Thus, these approaches do not scale well.

Correctness proofs. With correctness proofs, the difficulty is in constructing the proof. As with program synthesis, it is necessary to formalize a great deal of domain and programming knowledge in order for a proof to be constructed [6]. As the size of the program increases, the complexity of the proof increases. Automated theorem provers have exponential search space, and tools using this approach do not scale well. In practice, only critical, error-prone regions of code are proven correct using this technique.

Code analysis and reviews. Code reviews include code inspections and walkthroughs. These are manual, human-intensive activities. Studies show that these can lead to reduction in the number of errors in code by a factor of ten [11]. Code may be annotated to enhance inspections [44], or to facilitate the automatic derivation of verification conditions [2].

Testing. Testing is the most commonly used technique to verify systems. The difficulty with testing is the infeasibility of conducting exhaustive testing to guarantee correctness and the cost associated with extensive testing.

Model checking. Model checking [6,22,36] is a technique used to prove that a model satisfies a given property. It enables the correct construction of complex systems and has been quite successful in exposing errors in a variety of applications. Model checkers generally suffer from two difficulties: the creation of an accurate model, and the explosion in the state space that must be explored. System models can be represented by finite state-transition graphs; however, there is a limit to the number of states that can be handled. The state explosion problem arises since the number of states in the model being verified is exponential in the number of parallel components of the system [4].

3.2 Post-Deployment Techniques

Post-deployment verification techniques include multi-version programming and run-time monitoring. Each is discussed next.

Multi-version programming. Multi-version programming [29] involves writing multiple versions of the software and allowing the software to vote on the results during operation. In this approach, separate teams implement solutions for a single specification. It is assumed that the teams will develop different algorithms and that there is only a small probability of multiple teams introducing errors in the computation of a single function. Unfortunately, shared specifications can lead to common-cause failures. In addition, individuals tend to make mistakes in the more difficult parts of the problem rather than at random.

Run-time monitoring. With run-time monitoring, the program is examined as it executes to ensure that its behavior matches the expected behavior. Typically this is achieved by defining the expected behavior in terms of assertions associated with states of execution. There are numerous approaches to run-time monitoring [3,5,24,27,30,35,38,40]. These may be classified as hardware, software, and hybrid [39]. Hybrid monitoring, such as [20], brings together hardware and software monitoring. Hardware monitoring, such as [42,43], requires the use of a specialized hardware platform on which the application runs. In this setting, observer hardware monitors a program as it executes. This type of monitoring does not intrude on the program code; however, data is captured at a low level of abstraction making it difficult to interpret.

A software analog to hardware monitoring is to create an observer process that monitors a program as it executes. Here, the executing program sends state information to the observer process as it executes, and the observer analyzes the state information. A current example of this is [21,45]. A similar approach [27,28,26] is to use event specifications that relate state information to high-level events. In this approach, the observer communicates the information that is to be checked to a run-time monitor. Advantages of the observer architecture are that the changes to the original program are minimized, reducing the impact on program execution. Disadvantages are that the program must transfer enough state information to the observer in order for constraint checks to be run and it is more difficult for the observer to halt program execution in the event of a constraint violation.

Another approach to software monitoring is to insert code to test assertions into the program. The assertions are tested as the program executes. A common method [30,35,38] is for developers to manually create and insert constraint-testing code in program source code as programs are being developed. This process can be partially automated by annotating program source code with assertions and generating the assertion test at the point of annotation at compile time. Approaches that automate the determination of insertion points for constraint checking include [17,27]. More traditional approaches to software monitoring take cycles from an executing program to analyze state information.

A simple extension to run-time monitoring is post-mortem analysis. Post-mortem analysis [15,21] is similar to run-time monitoring in that sequences of states from a particular execution of a program are examined. In addition, as the program executes, information about the states visited by the program is stored. The sequence of state information is called an *execution trace*. After the program completes, the execution trace is analyzed. The advantages are that the program need not test constraints at runtime; thus, performance degradation is minimized, and some temporal constraints (e.g., property P holds until property Q becomes true) can be verified at any state in the execution trace. The disadvantage is that failures are not prevented from occurring.

4 Research Directions

The DynaMICs system is immature; however, based on our experiments, we feel that it shows promise as an extension to existing verification approaches. Parts of it have been prototyped and have been demonstrated on toy problems. It is our intent to improve this system and apply it to real-world software development efforts [16,34]. In order to do that, we have identified areas of future work. This section outlines the most important research and development: application of temporal logic, insertion of constraint-monitoring code for general programs, performance issues associated with monitoring, and soundness of the algorithms.

Application of temporal logic. Some practitioners may find expressing constraints in a formal temporal logic to be daunting. Our specification language is more intuitive, and we aim to show that our constraints may be expressed in a temporal logic and, thus, it may be possible to take advantage of reasoning systems that utilize temporal logic. Using the language defined in [8] event E and condition C in a DynaMICs specification can be expressed as: $\Box(E_s \rightarrow \bigcirc_s C)$, where E_s denotes an event that occurs in state s, $\bigcirc_s C$ denotes that condition C holds in the state immediately following s, and $\Box P$ denotes that P holds in the current and all future states. Automated reasoning is essential in DynaMICs for identifying inconsistencies, one of the main software-development problems addressed by the approach.

Performance. One major drawback to run-time constraint monitoring is the performance degradation that occurs. To address this, two approaches are being investigated. One is the identification of constraints that can be proven statically, reducing the number of constraints that need to be checked dynamically. The second approach is to offload at least some of the constraint monitoring to a co-processor. This co-processor will use a snoopy protocol to observe sequences of memory accesses and trigger constraint monitoring code when a store is made to a variable of interest [42]. Synchronization between the main processor and the co-processor will be necessary, particularly in the case of mission-critical constraints.

Insertion of constraint-monitoring code. The constraint monitoring code that we have been able to insert thus far has been tied to events that are changes in variables of interest. We have not addressed the issues of indirection, procedure call parameters, dynamic binding, or intermediate values stored only in registers (when examining object code). Static analysis alone may not be sufficient to determine appropriate monitoring points during execution. The snoopy/co-processor approach may be of value for catching indirect accesses to memory, since the co-processor will trigger constraint tests based on accesses to memory, whether made through direct or indirect reference.

It appears that there are useful constraints that require the monitoring of use as well as assignment. For example, consider the precondition, "Ignore output of drifting pressure transducers" [31]. There are two different uses of the transducer's output value. One of these is used to compute pressure. The other is used to determine if the transducer is drifting. The former is not allowed when the transducer is drifting, while the latter is allowed at any time. We have not yet considered monitoring use of variables of interest.

Soundness. There are two concerns with respect to soundness of the monitor. One is that the monitor detects a violation whenever one exists, and the other is that constraints are not checked where they may give rise to an erroneous detection. In other words, we want assurance that the monitor does not give rise to false negatives or false positives.

5 Summary

The following statement summarizes the state of the art of program verification: "We can't tell you how good it is, but we can tell you how hard we tried" [37]. Although there have been successes in proving correctness using formal methods, approaches such as theorem proving and deductive synthesis are not able to provide assurance that execution of code for a complex system is correct for all possible inputs. As a result, it is desirable to add safety features to programs in order to determine if a program behaves as expected. As with program verification using theorem proving, the DynaMICs approach shows correctness. Instead of proving validity, however, we are proving satisfiability of assertions for a particular interpretation of a model, i.e., the interpretation of the current program execution. We claim, that this is the most important interpretation.

The DynaMICs approach differs from other monitoring approaches in that constraints are elicited at all stages of the software development lifecycle and instrumentation of program code is automated. We believe that the automation provided by the approach, the ability to monitor correct behavior of programs, and the ability to trace to artifacts will motivate the capture and use of constraints. A tool that supports reasoning about constraints and detection of potential inconsistencies in requirements will make the approach even more attractive.

References

[1] Bertolino, A., and M. Marré, *Deriving Path Expressions Recursively*, Proceedings IEEE Second Workshop on Program Comprehension, 177–185, 1993.

[2] Bonnier, S., and T. Heyer, *COMPASS: A Comprehensible Assertion Method*, TAPSOFT '07: Theory and Practice of Software Development, Lecture Notes in Computer Science, 803–817, 1997.

[3] Bruegge, B., T. Gottschal, and B. Luo, *A Framework for Dynamic Program Analyzers*, Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, 65–82, 1993.

[4] Campos, S., and E. Clarke, *Analysis and Verification of Real-time Systems Using Quantitative Symbolic Algorithms*, Journal of Software Tools for Technology Transfer, **2**(1999).

[5] Chodrow, S., and M. Gouda, *Implementation of the Sentry System*, Software Practice and Experience, **25**(1995), 373–387.

[6] Clarke, E. M., J. M. Wing et al., *Formal Methods: State of the Art and Future Directions*, ACM Computing Surveys, **28**(1996), 626–643.

[7] DoD-Std-2167A. "Military Standard: Defense System Software Development," Department of Defense, Washington, DC., 1988.

[8] Dardenne, A., A. van Lamsweerde, and S. Fickas, *Goal-directed Requirements Acquisition*, Science of Computer Programming, **20**(1993), 3–50.

[9] Fernandez, F., "Compiler-driven Approach to Monitoring Integrity Constraints," Master's Thesis, The University of Texas at El Paso, El Paso, Texas, 1998.

[10] Flener, P., "Logic Program Synthesis from Incomplete Information," Klewer Academic Publishers, Norwell, MA, 1995.

[11] Freedman, D. P., and G. M. Weinberg, "Handbook of Walkthroughs, Inspections, and Technical Reviews, Evaluating Programs, Projects, and Products," Dorset House, New York, NY, 1990.

[12] Gates, A., *On Defining a Class of Integrity Constraints,* Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering, 338–344, 1996.

[13] Gates, A., and S. Li, *Software Faults and their Detection through DynaMICs*, Proceedings of the IASTED Conference Software Engineering, 323–327, 1998.

[14] Gates, A. Q., O. Mondragon, S. Roach, and A. Provetti, "Object-level Constraint Instrumentation: From Control Flowgraphs to Path Expressions," Technical Report, 2001.

[15] Gates, A. Q., and O. Mondragon, *A Constraint-Based Tracing Approach*, to appear Journal of Systems and Software, 2002.

[16] Gates, A. Q., and S. Roach, *A Framework for Knowledge Management and Automated Constraint Monitoring*, Proceedings of the Monterey Workshop 2001: Engineering Automation for Software Intensive System Integration, 154-168, 2001.

[17] Gates, A. Q., and P. J. Teller, *DynaMICs: An Automated and Independent Software-Fault Detection Approach*, Proceedings of the Fourth International High-Assurance Systems Engineering Symposium, 11–19, 1999.

[18] Gates, A. Q., and P. J. Teller, *An Integrated Design of a Dynamic Software-Fault Monitoring System*, Journal of Integrated Design & Process Science, **14**(2000), 63–78.

[19] Gates, A. Q., and P. J. Teller, "Dynamic Software Monitoring with Integrity Constraints: A Unified Approach for the Development and Evolution of Software," Technical Report, 2001.

[20] Haban, D., and D. Wybranietz, *A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems*, IEEE Transactions on Software Engineering, **16**(1990), 197–211.

[21] Havelund, K., *Using Runtime Analysis to Guide Model Checking of Java Programs*, The Seventh International SPIN Workshop, Stanford, California, 2000.

[22] Holtzmann, G., *The Spin Model Checker*, IEEE Transactions on Software Engineering, **23**(1997), 279–295.

[23] Horowitz, E., and R. Williamson, *SODOS: A Software Documentation Support Environment - Its Use*, IEEE Transactions on Software Engineering, **12**(1986), 1076–1087.

[24] Jeffery, C., W. Zhou, K. Templer and M. Brazell, *A Lightweight Architecture for Program Execution Monitoring*, ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 67–74, 1998.

[25] Kidd, M., *Ensuring Critical Event Sequences in High Consequence Computer Based Systems as Inspired by Path Expressions*, Proceedings of the International Conference and Workshop on Engineering of Computer Based Systems, 483–490, 1997.

[26] Kim, M., M. Viswanathan et al., "MaC: A Framework for Run-time Correctness Assurance of Real-Time Systems," Technical Report, 1998.

[27] Kim, M., M. Viswanathan et al., *Formally Specified Monitoring of Temporal Properties*, Proceedings of the European Conference on Real-Time Systems, 1999.

[28] Lee, I., H. Ben-Abdallah et al., *A Monitoring and Checking Framework for Run-time Correctness Assurance*, Proceedings of the 1998 Korea-U.S. Technical Conference on Strategic Technologies, 1998.

[29] Leveson, N. G., "Safeware: System Safety and Computers," Addision-Wesley, Reading, MA, 1995.

[30] Luckham, D., W. Mann, S. Meldal, and D. Helmbold, *An Overview of Anna: Specification Language for Ada,* IEEE Software, **20**(1988), 9–23.

[31] Lutz, R. R., and H. Shaw, "DS-1 Pilot Implementation of Assertions: Summary of Lessons Learned," Technical Report, Jet Propulsion Laboratory, Pasadena, CA, 1997.

[32] Manna, Z., and R. J. Waldinger, *A Deductive Approach to Program Synthesis*, ACM Transactions on Programming Languages and Systems, **2**(1980), 90–121.

[33] Pinheiro, F., A. Goguen, *An Object-Oriented Tool for Tracing Requirements*, IEEE Software, **13**(1996), 52–64.

[34] Roach, S., and A. Q. Gates, *Towards Synthesis of Runtime Constraint Monitoring Code*, Proceedings of the 2nd International Workshop on Automated Program Analysis, Testing, and Verification, 2001.

[35] Rosenblum, D.S., *A Practical Approach to Programming with Assertions*, IEEE Transactions on Software Engineering, **21**(1995), 19–31.

[36] Rushby, J., *Theorem Proving for Verification*, in "Modelling and Verification of Parallel Processes" (F. Cassez, ed.), Springer-Verlag, Nantes, France, 2000.

[37] Rushby, J., Keynote Speech, High Assurance Systems Engineering Symposium 2000, Albuquerque, NM, November 2000.

[38] Sankar, S. and M. Mandal, *Concurrent Runtime Monitoring of Formally Specified Programs*, IEEE Computer, **26**(1993), 32–41.

[39] Schroeder, B. A., *On-line Monitoring: a Tutorial*, IEEE Computer, **28**(1995), 72-78.

[40] Sosic, R., *Dynascope: A Tool for Program Directing*, Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation, 12–21, 1992.

[41] Tarjan, R. E., *Fast Algorithms for Solving Path Problems*, Journal of the ACM, **28**(1981), 584–614.

[42] Teller, P., M. Maxwell, and A. Gates, *Towards the Design of a Snoopy Coprocessor for Dynamic Software-Fault Detection*, Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference, 310–317, 1999.

[43] Tsai, J.P., K. Fang, H. Chen, and Y. Bi, *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging*, IEEE Transactions on Software Engineering, **16**(1990), 897–916.

[44] van Emden, M. H., *Structured Inspection of Code*, Software Testing, Verification and Reliability, **2**(1992), 133–153.

[45] Visser, W., K. Havelund et al., *Java PathFinder - Second Generation of a Java Model Checker*, Post-CAV Workshop on Advances in Verification, July 2000.