A Framework for Knowledge Management and Automated Constraint Monitoring

Ann Q. Gates and Steve Roach
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas 79968
agates, sroach@cs.utep.edu

Abstract

This paper describes an approach called Dynamic Monitoring with Integrity Constraints (DynaMICs) that consists of a specification language for defining constraints and tools that permit automated instrumentation of constraints, runtime monitoring that minimizes performance degradation, and tracing. The goal is to capture domain and system knowledge as constraints and to use the constraints to monitor software execution, providing evidence of correctness and assistance in identification of error sources. The paper presents a framework for managing knowledge and instrumenting programs to test the state of programs at runtime. In addition, it discusses the role that temporal logic, model checking and program-synthesis systems can play in developing and using DynaMICs.

1. Introduction

The successful development of complex software systems typically requires management of two types of knowledge: domain knowledge and system knowledge. Domain knowledge is that knowledge needed to understand problems and solutions in a particular area of human endeavor. For example, in order to construct a program that simulates wind tunnel experiments, system developers would need knowledge of fluid dynamics. Domain knowledge is elicited from domain experts, customers, users, and possibly from written reference material. System knowledge is that knowledge needed to understand the design and implementation of a software system that solves a problem from a particular domain. For example, in order to implement a module for an information management system, system developers would need to understand the data structures used in programs and how they map to the application domain. System knowledge comes from designers, implementers, and maintainers.

The integration of domain and system knowledge is essential for successful software development. This includes the capture and communication of such knowledge among team members throughout the software lifecycle. In addition, understanding the relations between domain and system knowledge can help close the gap between a software failure and the software fault that leads to that failure.

One technique for capturing domain and system knowledge is through the use of integrity constraints. *Integrity constraints*, referred to as *constraints* in the remainder of this paper, are propositions about the state of a system. *State* is defined as a set of program-variable and value pairs that captures a

snapshot of memory during program execution. Constraints can be used to check during runtime that the system is meeting its requirements and that assumptions and limitations, which arise from design and implementation decisions, hold. Not only can constraints be used to identify errors and assist in debugging, they also can provide evidence of correctness at execution time, maintain knowledge about the domain and system, and assist in planning and implementing maintenance of the software.

This paper presents a framework for managing domain and system knowledge and for instrumenting programs to test the state of programs at runtime. The ultimate goal of this work is to automate the insertion of constraint-checking code in order to facilitate the use of constraints. With automated instrumentation, domain experts and system implementers may be more willing to expend effort generating the needed constraints to ensure correct execution.

1.1 DynaMICs

Dynamic Monitoring with Integrity Constraints (DynaMICs) is an approach that captures domain and system knowledge through constraints to ensure the correct functioning of a program during its execution (GT99, GT00, GT01). Figure 1 presents a high-level view of the DynaMICs approach. The main features that differentiate DynaMICs from other software-fault monitoring approaches are that constraint specifications are maintained in a repository separate from other artifacts, and constraint-checking code is automatically inserted into the code. The separation of constraint specifications from code facilitates identification of potential conflicts among constraints throughout the software's lifecycle. Algorithms translate propositions into constraint-checking code and determine the execution points at which the constraint-checking code is to be embedded into the program.

The tracing mechanism (GM01) provides support for establishing linkages between constraints and artifacts, which along with linkages that are created automatically during instrumentation, permits the following types of tracing: from application source code to constraint, from constraint to application source code, from constraint to artifacts, and from application code to artifacts. Artifacts include requirements. Because the links between the application code and constraints are created automatically during the instrumentation process, the links are maintainable. In addition, it eliminates the need for physical links between artifacts and application code, which relieves the programmer from managing links in the code when code is revised, a tedious task that is prone to error. The approach addresses some of the issues that have slowed extensive adoption of tracing. Specifically, DynaMICs targets simplified tracing of constraints among documents, reduced overhead for tracing and, in conjunction with monitoring, requirements compliance with respect to constraints.

One of the reasons for the lack of widespread adoption of runtime monitoring is the performance degradation that results when constraint checks have been inserted into the program code. To address this,

several different types of monitors are being investigated, e.g., one in which monitoring responsibilities are delegated to a process other than the one executing the application program (TM99, SM93).

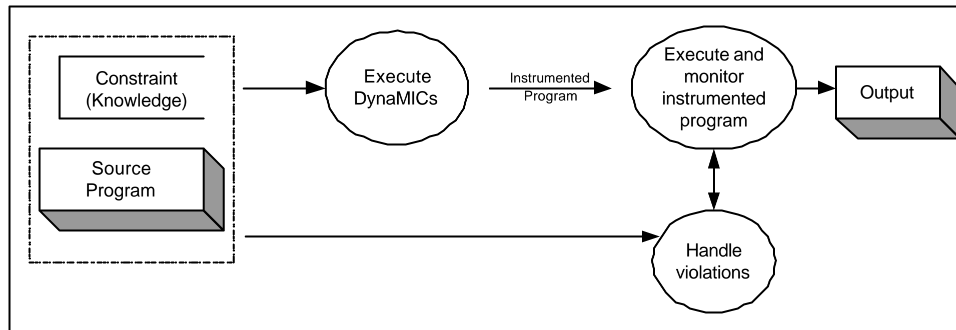

Figure 1: A high-level view of DynaMICs.

1.2 Impact of DynaMICs

Determining the impact of change on a system is difficult. In particular, tools are needed to facilitate the identification of conflicts when requirements change. By managing domain and system knowledge, DynaMICs provides such support. Formally capturing knowledge allows reasoning about the effects of change and reduces the probability of creating errors in the software. The types of faults that are detectable through DynaMICs are dependent on the constraints specified by the development team. DynaMICs targets requirements faults, in particular those resulting from incorrect, inconsistent, and ambiguous requirements (GL98). In addition, DynaMICs is effective at detecting when an unpredictable sequence of events results in inconsistent data in the system (C01) and when the context of the program's operation changes as described in (LC95).

Communicating information among team members and eliminating conflicts in requirements are major concerns during development of complex systems (CK88, DV93). While successful software development efforts typically include at least one person who can integrate different perspectives on the development process as well as domain and system knowledge, there is an inherent risk in such dependence on key personnel. In addition, there is a limit as to how much knowledge can be managed by a single person, especially as systems become larger, more complex, and cross application domains. The capture of critical knowledge through constraints and the separation of constraints from the source program assist information management and communication.

Automated instrumentation of program code provides assurance of constraint coverage, i.e., DynaMICs determines the points in the program at which constraints from the repository should be checked. It also simplifies maintenance of constraints because changes are made to high-level constraint

specifications and not to the code itself. This approach reduces the number of errors that can be introduced during maintenance because it is not possible for insertion or deletion of program code to corrupt previously inserted monitoring code. By establishing relations between constraint specifications and supporting documentation, it is also possible to provide justification of constraints. This supports resolution of conflicts between specifications and code.

An area in which this work can have impact is in creation and execution of test beds. Adding constraints during testing typically requires manual instrumentation of checking code and removal of the code when the product is deployed. Additionally, because of the large increase in code size due to instrumentation, execution times are naturally increased. By automating the insertion of constraint-checking code, developers are free from the task of instrumentation and can focus instead on improving the knowledge base that indicates correct program execution. Defects can be identified earlier in the testing cycle, reducing cost.

1.3 Organization of Paper

The remaining sections in this paper examine the constraint specification and automated instrumentation components of DynaMICs in which formal methods are critical. The main points covered in each section are as follows:

- overview of component, including a description of the knowledge needed, where it is obtained, and how its used;
- approach used to realize the component, including a discussion of the technologies that have been used to develop a proof-of-concept; and
- integration of formal methods, including a discussion of formal methods that impact development of a runtime monitoring system such as DynaMICs.

In addition, the paper summarizes related work.

2. Constraint Specification

2.1 Overview

Initial work on DynaMICs has focused mainly on capturing domain and system knowledge through constraints. To have a larger impact on system development and maintenance, the knowledge base of DynaMICs must be extended to include additional system knowledge, in particular design knowledge derived from objects, data structures, operations, relationships among operations, and algorithms. Nevertheless, constraint-checking code establishes an implicit relationship between domain knowledge and the system.

Domain experts, clients, users, and members of the development team contribute to constraint definition. This information may be traced to interview transcripts, memoranda, reports, and the requirements specification (GK97, DV93). Members of the development team contribute constraints by applying assumptions about the operating environment (e.g., acceptable input values) and limitations imposed by the design (e.g., size of a data structure). Testers, who are interested in monitoring program behavior under specific testing conditions, can add special-purpose constraints.

Constraint elicitation and identification necessitates analysis of the problem from a perspective different than requirements analysis. Requirements answer the question, "What will the system do?" while constraints answer the question "What monitored relationships can indicate correct program execution?". An example is presented in Section 4. A more detailed description of the constraint-definition process can be found in (GM01).

2.2 Approach

Constraints specifications consist of three parts: events, conditions, and actions. The event directs instrumentation by specifying when the condition must hold. The condition specifies the constraint, and the action specifies what must be performed on violation of the condition. Each is discussed in the subsections that follow.

Constraints are captured during the requirements elicitation, requirements analysis, design, and implementation phases of software development. As a result, it is necessary to maintain the mapping from terms in the constraint specification to variables and storage locations at the program-code level. This is done through a data dictionary that maintains information about variables used in specifications, called *constraint variables*. Program development personnel maintain the associations of the constraint variables to program variables during program construction.

2.2.1 Event Specification

The event definition directs program instrumentation and is defined as an ordered four-tuple (GT99):

Event : Variable-set × Transition × Phase × Placement

Variable-set	: set-of-tokens
Transition	: {immediate, intermediate, delayed}
Phase	: {input, processing, output}
Placement	: {before-store, after-store}

Events are based on stores to variables associated with constraint variables. *Variable-set* maintains the set of constraint-variable names for which *state transitions* (a change in the values held in constraint variables) are observed. Constraint variables that refer to values from previous states are distinguished. The name *static constraint* is used to refer to constraints that are monitoring only the current state of a

constraint variable. *Transitional constraint* refers to a constraint that refers to initial or previous state of constraint variables.

Variable-set, Transition, and *Phase* identify the state transitions to be monitored with respect to a specified phase. Valid phases include *input, processing,* and *output.* For a specified phase and *Variable-set, immediate* 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 *delayed-on-input* constraint for variables *a, b,* and *c* 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. For the specified phase and *Variable-set*, an *intermediate* value designates that the constraint must hold after an implied sequence of state transitions. For example, if *Variable-set* contains variables *a* and *b* and both variables are updated in a sequence, then the monitoring code executes after the sequence completes.

The types of instructions that cause state transitions within a program and that can be used to determine potential points of program instrumentation include input, assignment, and output instructions, i.e., instructions that store data to memory. Each is associated with a computation phase. 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. An assumption is made that controls of sensitive devices are memory mapped and, thus, are accessed using assignments. Constraints on file output, with a format that is clearly specified, can be checked (GT01).

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. A *critical* constraint is one in which a violation of the constraint indicates a hazard condition that could result in catastrophic failure. Constraints for critical and non-critical constraints can be checked after a store to a variable.

2.2.2 Condition Specification

The condition definition is expressed in a first-order language (G96). Section 4 discusses the types of constraints that can be expressed in DynaMICs. Because constraints are implementation independent, a data dictionary, as described earlier, maps variables used in the constraint specification to program variables in the implementation and describes the real-world objects being modeled.

2.2.3 Action Specification

The action specification defines 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, and initiating graceful degradation. The latter three actions are currently under study.

2.2.4 Specification of Additional Information

Constraints may require information that is not computed by the program. This includes information that can be inferred and computed, e.g., using counters and accumulators. The collection of this information is defined using an event-condition-action specification, where the event and condition are the same as a constraint specification. The main difference is that the action, whic h computes values to be used by constraints, is triggered when the condition is satisfied. Consider a constraint that requires that *a* be less than *b* after a specified number of updates to *b*. In this case, variable, *c* is introduced to maintain the count of the number of updates to *b*. The event is an assignment to *b* (immediate-on-processing), the condition is *true,* and the action is the increment of *c*. The constraint can be specified as $c > threshold$ ® $a < b$ and is classified as an immediate-on-processing constraint on variable *b*.

2.3 Integration of Formal Methods

Some practitioners may find expressing constraints in a formal temporal logic to be daunting. We believe that our specification language is more intuitive and can be expressed in a formal temporal logic. Using the language defined in (DV93), event *E* and condition *C* in a DynaMICs specification can be expressed as: $\Box(E_s \rightarrow O_s C)$, where E_s denotes an event that occurs in state *s*, O_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. Clearly, the advantage of a formal language is the ability to reason about constraints. This is essential for dealing with inconsistencies, one of the main software-development problems addressed by DynaMICs.

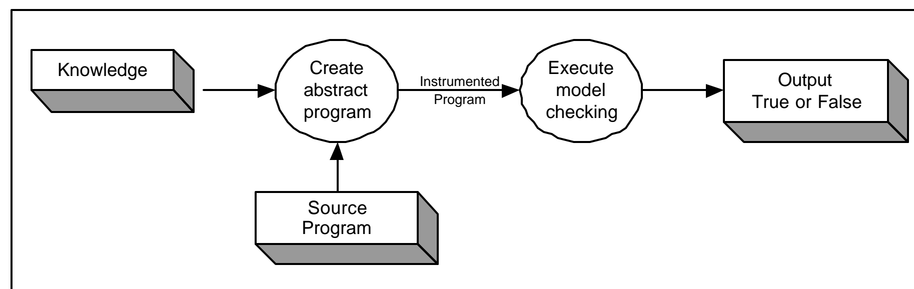

Figure 2: A high-level view of model checking.

The goals of runtime constraint checking are similar to model checking (H97). Model checkers monitor constraints of programs using an exhaustive finite-state search of an abstracted program (see Figure 2). Because the state space of complex programs is too large for exhaustive search, a common approach is to abstract the program to a model and test properties of the model. DynaMICs extends the range of model checking by allowing verification of constraints over a larger state space. The approach provides runtime assurance that a property holds in a particular execution or test suite, i.e., it explores only a subset of the state space that a program visits and not the entire state space. Model checking will detect a failure if an error exists in the model. On the other hand, DynaMICs will detect a failure if an error exists in the program on a tested execution path. Model checking verifies an abstraction of a real program. DynaMICs verifies a subset of states of a program.

One of the difficulties in model checking is the correct construction and instrumentation of a model from a program. We would like to investigate the possibility of automating model instrumentation for verification via model checking. Another area of research is the vertical traceability of errors detected by a model checker to code and specifications. One possible approach is to identify a failure using a model checker, instrument a source program using DynaMICs, then execute the instrumented source code using the state trace from the model checker to build a test suite. DynaMICs has the capability to trace to artifacts upon constraint violations through the tracing mechanism.

3. Instrumentation

3.1 Overview

Monitoring code is code that results in the test of a constraint. This can be an in-line sequence of instructions, a call to a procedure or function, or a trigger for an instruction sequence executed in a separate process (F98, L95). Identifying the points in program execution at which monitoring code should be executed requires the event definition of a specification along with analysis of the program's control flow. This section examines control-flow graph analysis and code generation.

3.2 Approach

3.2.1 Control-Flow Analysis

Analysis of a program's execution flow is needed in order to automatically determine program instrumentation points from constraint specifications (GP01). These points are associated with updates to monitored variables. The analysis approach followed by DynaMICS focuses on *path expressions* (T81, BM93, K97), i.e., regular expressions derivable from execution control-flow graphs (CFGs), each node of which is a basic block. A *basic block* (AS86) is a sequence of instructions with a single entry and single exit. Read/write lists are associated with each basic block to identify basic blocks of interest, i.e., ones

that include accesses to monitored variables. Once these basic blocks are identified, the path expressions can be condensed by coalescing adjacent basic blocks.

Using the event definition as a guide, path expressions are analyzed to identify program instrumentation points; each of these points is associated with a unique *path tag* (TM99). Each path tag maps to a constraint specification. An algorithm for defining path tags for immediate, delayed-on-input, delayed-on-processing, and delayed-on-output constraints can be found in (F98, GT99, GP01). Checking immediate constraints is straightforward; the constraint check is performed whenever a monitored variable is modified. Placement of a delayed constraint requires identifying the best location at which to place a constraint check. For a delayed-on-process constraint, this is the point where assignment of the monitored variables is complete.

Analysis can be done at the intermediate-code or object-code level. In the case of mission-critical constraints, object-code analysis is needed to prevent a transition to an unsafe state. In memory-mapped IO systems, it may be necessary to prevent a write to memory prior to testing a constraint. For constraints that are not critical, intermediate-code instrumentation is sufficient. Because safety-critical systems require assurance that is not provided by current compilers, checking of critical and mission-critical constraints requires instrumentation at the object-code level.

3.2.2 Code Generation

One goal of DynaMICs is to synthesize constraint-checking code automatically from constraint specifications. Two classes of code are considered: *constraint-checking code* which ensures that the monitored program is executing correctly, as defined by constraints; and *information-generating code* which computes additional information needed to check constraints. Only the constraint-checking code raises violations for handling by the monitor. Neither will alter execution of the program except in the case when a violation is detected by constraint-checking code and the corresponding action definition requires error recovery. The information collected by information-generating code does not affect program execution since the variables used to maintain the information cannot be referenced in the source code. Because constraint specifications and domain knowledge are implementation independent, the code-generation algorithm needs to translate specifications to code, considering possible differences between constraint-variable data types and associated program-variable data types stored in the data dictionary.

3.3 Integration of Formal Methods

Examples of program-synthesis systems that generate concrete-level code from abstract-level specifications are known (S91, SW94, SM96, W99). Some of these systems, in particular the fully automated deductive systems, suffer from their dependence on automated-theorem-proving tools.

However, it is possible that most of the constraints derived in practice will be classified by their structure (G96). In this case, it might be possible to take advantage of transformations, proof planning, or schema-guided synthesis (WB92, BS90). Experiments conducted thus far have shown that the code to test a constraint is reasonably small, on the order of tens of lines of high-level code (C01). This small size of code enables the use of fully automated tools.

The aforementioned approaches require complete axiomatization of domain knowledge. If such knowledge is not available, inductive techniques for the synthesis of constraint-checking code could be useful (F95, MB98). This would allow the collection of examples from which the system will be able to generalize. While these approaches require human interaction, it is possible to reuse the results in later syntheses.

4. An Example

Typical conditions that would be expressed by constraints include verification that values of a program variable belong to a specified set of values. Constraints can be used to check a count, summation, and maximum (minimum) of a collection of values associated with program variables, or determine that pertinent program variables have been assigned values prior to use. Other types of constraints include checking relationships between variables, e.g., inlet temperature must be less than the outlet temperature. In addition, constraints can capture set-membership properties, e.g., no part-time employee is a full-time employee (the set intersection is empty). Constraints examine program variable properties at different points in program execution. The interest is in examining intermediate values in addition to initial and final values of program variables. The work has investigated general constraints on toy problems that included text processing, transaction-oriented, graph manipulation, pattern matching, and real-number calculations.

To illustrate the specification, analysis, and instrumentation of constraints, a simple example is given. This example computes the division of two integers, a and b, yielding a quotient, q, and a remainder, r. For this problem, two constraints can be defined:

$$r < b \text{ and } (q \times b) + r = a.$$

The constraints do not recalculate the division of a and b, rather they check that the division is correct. Domain (division) expertise is required to specify constraints such as these.

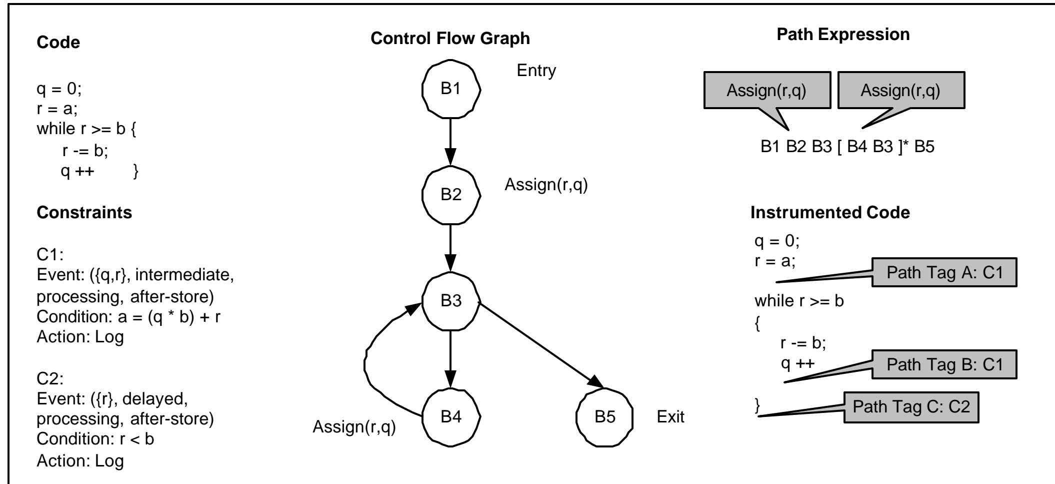

Figure 3: Constraint instrumentation for the division example.

5. Related Work

Approaches that provide evidence of correct program execution can be partitioned into those that are applied during software development and those that are applied after the product is deployed. The approaches that fall into the former partition include program synthesis, correctness proofs, code analysis, reviews, testing, and model checking. The approaches that fall into the latter partition include multi-version programming and runtime monitoring.

Monitoring can be done at the hardware- or software-level (S95). Early approaches to software-fault monitoring include Anna (L85), Concurrent Anna (S93), Annotation Pre-Processor for C program (APP) (R95), and Eiffel (M92). Each approach requires the developer to specify and insert assertions at appropriate locations in the application code. Assertions are a subclass of integrity constraints because integrity constraints extend the application of assertions by capturing knowledge that is implementation independent. Assertions, like integrity constraints, oversee the dynamic execution of programs by monitoring program variables.

Anna and Concurrent Anna use constructs to specify assertions on Ada programs, using formal comments that consist of Boolean-valued expressions and reserved words. In addition, the language allows specification of computations that are needed only to check assertions. In both approaches, formal comments are replaced by calls to appropriate checking modules. Concurrent Anna addresses performance degradation by providing concurrent execution of checking modules.

The APP approach incorporates assertions, written in an extension of C's expression language, to monitor C programs. As with Anna, the programmer inserts assertions at points in the program where checks should occur and at the module level through pre- and post-conditions. The compiler pre-processor pass is used to instrument checking code into the program.

Eiffel provides an object-oriented language, method, and environment. Unlike the previous approaches, Eiffel provides methodologies to support the analysis, design, and implementation phases of object-oriented software development. The **assert** macro/function used in various languages such as the C and C++ programming languages also provides monitoring capabilities. The **assert** macro/function typically is used by programmers to check pre- and post-conditions of code sections. If the **assert** function evaluates to false, a specified error message is output.

In contrast to the aforementioned approaches, Monitoring and Checking (MaC) architecture (LB98) does not provide software-fault monitoring through user-inserted assertions at the implementation (language) level, but rather through event specifications at the design level. MaC supports run-time monitoring via a run-time checker that interacts with filter and event recognizer components. During the design phase, the user creates a monitoring script that defines events, produces an event recognizer that relates state information to high-level events, and instruments the code that retrieves relevant program state. The filter component is responsible for extracting program variable values from system code and sending them to the event recognizer. The event recognizer communicates the information that is to be checked to the run-time checker. MaC targets detection of faults in numerical computations and sequencing of events. Other approaches to software-fault monitoring include (CG95, BT93).

6. Summary

The difficulty in communicating crucial knowledge among team members from specification to software deployment, managing change, and formally specifying requirements makes verification of software challenging. To address this, it is imperative that developers begin to focus on identifying pertinent domain and system knowledge that can be used to determine whether a program is operating correctly during its execution. Successful software projects have key personnel who can integrate several knowledge domains as well as system knowledge. The aim of the work presented in this paper is to capture such knowledge as constraints and to use the constraints to monitor the program during runtime. This will facilitate identification of errors by closing the gap between a software failure and the software fault that leads to failure. The focus of this paper is to describe a framework for an approach called DynaMICs that assists in providing evidence of correctness in software systems and assistance in identification of error sources. Additionally, the paper describes the practical impact of formal methods on development of DynaMICs.

The DynaMICs approach differs from other monitoring approaches because constraints are stored separately from other artifacts and instrumentation of constraint-checking 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. Because the formal language

used by DynaMICs may be expressed in a formal temporal logic, a tool that supports reasoning about constraints and detection of potential inconsistencies in requirements will make the approach even more attractive.

The goals of DynaMICs are complementary to model checking. The automated instrumentation algorithms of DynaMICs may be applicable to instrumentation of abstract programs in model checkers, and model checkers can support the use of DynaMICs. For example, model checkers can direct creation of test suites from state traces upon failures and, using these test suites, DynaMICs can facilitate the identification of faults in the program and assist in error resolution through the tracing mechanism.

In order for DynaMICs to be applicable to high-assurance systems, it is crucial to generate provably correct, executable code. Generation of constraint-checking code from specifications is a good candidate for application of program-synthesis systems that take advantage of proof-planning or schema-guided synthesis. The main reasons are the simple and concise pieces of code that are generated from each constraint and the fact that constraints can be classified based on their structure.

Acknowledgements

This work was sponsored by NASA grants NAG-1138, NAG-1012, and NCCW-0089, and NSF grants EIA-9522207 and EIA-9729990.

References

[AS86] Aho, A., Sethi, R. and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[BM93] A. Bertolino and M. Marré, "Deriving Path Expressions Recursively," *IEEE Second Workshop on Program Comprehension*, pp. 177-185, July 1993.

[BT93] B. Bruegge, T. Gottschalk 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.

[BS90] A. Bundy, A. Smaill, and G. Wiggins, "The Synthesis of Logic Programs from Inductive Proofs," in J. W. Loyd (ed), *Proceedings of the ESPRIT Symposium on Computational Logic*, pp. 135-149, Springer-Verlag, 1990.

[C01] R. Cereceres, "A Study Of The Effectiveness Of Integrity Constraints in the DynaMICs Approach", Master's Project Report, The University of Texas at El Paso, El Paso, Texas, 2001.

[CG95] S. Chodrow and M. Gouda, "Implementation of the Sentry System," Software-Practice and Experience, 25(4), 373-387, 1995.

[CK88] B. Curtis, H. Krasner, and N. Iscoe., "A Field Study of the Software Design Process for Large Systems", *Communications of the ACM*, 31(11), pp. 1268-1287, 1988.

[DV93] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed Requirements Acquistion", *Science of Computer Programming*, 20, pp. 3-50, 1993.

[F98] F. Fernandez, "Compiler-driven Approach to Monitoring Integrity Constraints", Master's Thesis, The University of Texas at El Paso, El Paso, Texas, 1998.

[F95] P. Flener, “Logic Program Synthesis from Incomplete Information”, Klewer Academic Publishers, Norwell, MA, 1995.

[G96] A. Gates, “On Defining a Class of Integrity Constraints”, *Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering*, pp. 338-344, 1996.

[GK97] A. Gates and C. Kubo Della-Piana, “The Identification of Integrity Constraints in Requirements for Context Monitoring”, *Proceedings of the International Conference and Workshop on Engineering of Computer-Based Systems*, pp. 498-505, 1997.

[GL98] A. Gates and S. Li, “Software Faults and their Detection through DynaMICs,” *Proceedings of the IASTED Conference Software Engineering*, Las Vegas, NV, pp. 323-327, 1998.

[GT99] A. Q. Gates and P. J. Teller, “DynaMICs: An Automated and Independent Software-Fault Detection Approach”, *Proceedings of the Fourth International High-Assurance Systems Engineering Symposium,* pp. 11-19,1999.

[GT00] A. Q. Gates and P. J. Teller, “An Integrated Design of a Dynamic Software-Fault Monitoring System”, *Journal of Integrated Design & Process Science. Society for Design and Process Science*, 14(3), 63-78, 2000.

[GT01] A. Q. Gates and P. J. Teller, “Dynamic Software Monitoring with Integrity Constraints: A Unified Approach for the Development and Evolution of Software”, in review, 2000.

[GP01] A. Q. Gates, O. Mondragon, S. Roach, and A. Provetti, “Object-level Constraint Instrumentation: From Control Flowgraphs to Path Expressions”, submitted to The Eighth International Static Analysis Symposium, February 2001.

[GM01] A. Q. Gates and O. Mondragon, “A Constraint-Based Tracing Approach”, to appear *Journal of Systems and Software,* 2001.

[H97] G. Holtzman, “The Spin Model Checker,” *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997.

[K97] M. Kidd, “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*, pp. 483-490, 1997

[L95] J. R. Larus, “EEL: Machine-Independent Executable Editing”, *SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* June 1995.

[LB98] I. Lee, 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.

[L85] D. Luckham, W. Mann, S. Meldal, and D. Helmbold, “An Overview of Anna: A Specification Language for Ada, ” *IEEE Software*, Vol. 20, No. 2, pp. 9-23, 1988.

[LC95] Luqi and D. Cooke, “How to Combine Nonmonotonic Logic and Rapid Prototyping to Maintain Software,” *International Journal on Software Engineering and Knowledge Engineering*, 5(1), pp. 89-118, 1995.

[M92] B. Meyer, *Eiffel: The Language,* NY: Prentice Hall, 1992.

[MB98] S. Muggleton and W. Buntine, “Machine invention of first-order predicates by inverting resolution,” *Proceedings of the 1988 International Conference on Machine Learning*, pp. 339-352, 1988.

[R95] D. S. Rosenblum, “A Practical Approach to Programming with Assertions,” *IEEE Transactions on Software Engineering*, 21(1), 19-31, 1995.

[SM93] S. Sankar and M. Mandal, “Concurrent Runtime Monitoring of Formally Specified Programs,” *IEEE Computer*, 26(3), pp. 32-41, 1993.

[S95] B.A. Schroeder, “On-line Monitoring: a Tutorial,” *Computer*, 28(6), 72-78, 1995.

[S91] D. R. Smith, "KIDS: A Knowledge-Based Software Development System", in *Automating Software Design*, M. Lowry and R. McCartney (eds.), MIT Press, pp. 483-514, 1991.

[S93] S. Sankar and M. Mandal, "Concurrent Runtime Monitoring of Formally Specified Programs," *IEEE Computer*, 26(3), 32-41, 1993.

[SM93] Y. V. Srinivas and J. L. McDonald, "The Architecture of Specware, a Formal Software Development System", Kestrel Institute Technical Report KES.U.96.7, 1996.

[SW94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood, "Deductive Composition of Astronomical Software from Subroutine Libraries", *Proceedings of the 12th Conference on Automated Deduction*, Nancy, France, June 28-July 1, 1994.

[T81] Tarjan, R.E., "Fast Algorithms for Solving Paths Problems," *Journal of the ACM*, 28(3), pp 584-614, 1981.

[TM99] P. Teller, 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*, pp. 310-317, February 1999.

[WB92] G. Wiggins, A. Bundy, I. Kraan, and J. Hesketh, "Synthesis and Transformation of Logic Programs from Constructive, Inductive Proof," *Proceedings of LOPSTR'91*, Springer-Verlag, pp. 27-45, 1992.

[W99] V. Winter, "An Overview of HATS: A Language Independent High Assurance Transformation System", Sandia National Laboratories, 1999.