

Instrumentation of Intermediate Code for Runtime Verification

Ann Quiroz Gates, Oscar Mondragon, Mary Payne, Steve Roach
The University of Texas at El Paso
Department of Computer Science
oscar, agates, mpayne, sroach@cs.utep.edu

Abstract

Runtime monitoring is aimed at ensuring correct runtime behavior with respect to specified constraints. It provides assurance that properties are maintained during a given program execution. The Dynamic Monitoring with Integrity Constraints (DynaMICs) approach is a runtime monitoring system under development at the University of Texas at El Paso. The focus of the paper is on the identification of instructions at the object-code level that require instrumentation for monitoring. Automated instrumentation is desirable because it can reduce errors introduced by humans, it provides finer control over instrumentation, and it allows greater control over instrumentation. The paper also discusses two other technologies associated with DynaMICs: the elicitation and formal specification of properties and constraint; and tracing property or constraint violations to the software engineering artifacts from which the constraints and properties were derived.

1. Introduction

Reliance on computer systems, particularly those in safety critical or mission critical systems, requires that these systems behave in a predictable manner. Traditional verification approaches such as testing and inspections are not sufficient to ensure the correct behavior of software, especially in the presence of subtle faults. In the May 2002 RTI Planning Report to the National Institute of Standards and Technology, it is reported that testing comprises 30%-90% of labor expended to produce a working program [RT02]. Nonetheless, 5% of all errors found during the lifetime of the product are discovered after product release. In 1999, the estimated cost due to software error in the aerospace industry alone was \$6 billion. In spite of the current best practices [CW96, HM96, H97, RJ00], significant and costly errors remain in production software.

1.1. Runtime monitoring

Runtime monitoring is aimed at ensuring correct runtime behavior with respect to specified constraints. It provides assurance that properties are maintained during a given program execution. Dynamic software-fault

monitoring approaches address reliability, safe failure, and avoidance of hazardous states with respect to software systems. Properties are verified at runtime to provide an additional layer of defense against catastrophic software failure. These techniques are used to identify software faults that were not detected during the verification and validation process and to improve the efficiency of these processes.

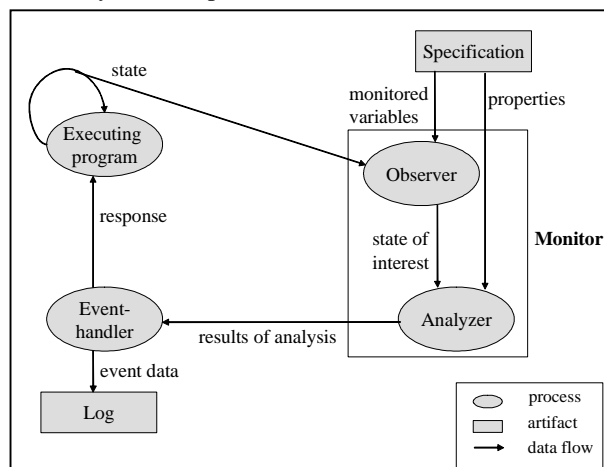


Figure 1: Runtime Monitoring Systems

Fig. 1 presents a high-level view of a runtime monitor. A runtime monitor is composed of two parts: an *observer* that evaluates a condition on a program state and an *analyzer* that evaluates an assertion. For example, suppose a program must never allow two processes access to a critical region simultaneously. This property may be specified for a runtime 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 when a process enters the critical region. When this event occurs, the analyzer tests the proposition that the number of processes in the critical region is exactly one. If the analyzer detects a violation of a constraint, the runtime monitoring system must respond in some fashion, e.g., by logging the occurrence, halting the program, or entering a recovery routine. The mechanism that responds to violations is referred to as an *event-handler*.

1.2. DynaMICs

The Dynamic Monitoring with Integrity Constraints (DynaMICs) approach is under development at the University of Texas at El Paso [GR01]. In DynaMICs, property and constraint specifications describing the behavior of an application are maintained separately from the program. The approach provides the facility to trace these specifications to software engineering artifacts such as an SRS, interview reports, and memoranda [GM02]. In addition, the formal specifications are translated into executable code segments, and programs are automatically instrumented at the object or byte code level. Properties are elicited during all stages of the development cycle. This allows domain-specific properties to be elicited from domain experts, design-specific properties to be elicited from system designers, and implementation-specific constraints to be elicited from programmers and developers. Properties can also be used to capture assumptions and limitations imposed on the application by the design and implementation.

The main advantage of separating properties from the program is that properties elicited from domain experts can be captured early in the lifecycle. Such properties frequently can be reused. Furthermore, delaying property specification to the implementation phase can result in domain-specific properties being overlooked.

A data dictionary is used to map variables used in properties to variables used during implementation. This mapping, along with a constraint's specification, is used to generate checking code.

DynaMICs automatically instruments program code [GR01]. Automated instrumentation makes runtime monitoring cost effective by reducing the expense of maintaining instrumented code and ensuring that instrumentation is accurately implemented. This paper presents an overview of the salient features of DynaMICs, focusing on techniques for identifying points in intermediate- or object-level code where constraints must be tested at runtime.

2. Specification and Tracing in DynaMICs

2.1. Specification of Program Properties

Constraints in DynaMICs are specified as event-condition-action rules, where an event initiates evaluation of a condition that guards an action [GR01]. DynaMICs initiates property checks when state transitions occur due to modifications of monitored variables. Modifications occur because of an input, assignment, or output statement. DynaMICs events are defined as ordered tuples over (*Variable-set*, *Phase*, *Transition*).

Variable-set contains all of the variables that are subject to monitoring with respect to a specific property.

A property relates variables to the behavior of real-world objects. For example, suppose that during the simulation of an engine, variable T represents the temperature of the engine block. If a property is specified on engine block temperature, then a store to T is the event that triggers the checking of this property. For this property, the *variable-set* contains T .

Valid phases are *input*, *processing*, and *output*. The input phase begins at the instruction when data is first acquired from a source external to the program and ends with the last such instruction. The output phase begins with the execution of the first output statement and ends with the last such statement. The processing phase is the entire program. These phases may overlap.

For a specified phase and variable set, an *immediate* transition indicates that the constraint must hold after each state transition associated with a phase. *Delayed* denotes that a constraint must hold at the end of a specified phase. For example, a *delayed-on-input* constraint for a set of variables indicates that the associated monitoring code executes after all values for these variables have been read. The observer can qualify the events that are monitored, extending the type of monitoring that can be accomplished through this approach.

2.2. Elicitation and Specification via PROSPEC

Prospec is a tool that assists practitioners in the elicitation and specification of system properties. Prospec is built upon the Specification Pattern System (SPS) [DA98] and uses property patterns and scopes. Prospec extends SPS through *composite propositions*, which characterizes sequential and concurrent behavior for multiple propositions. Multiple conditions or events may represent behavior such as sequences, concurrency, and non-determinism and may define the boundaries of scopes or type of patterns. As an elicitation approach, composite propositions can highlight subtle details of sequences and concurrent behavior by using directed questions to guide the practitioner. Resolving the choices addressed by these questions results in a precise specification of behavior associated to multiple conditions or events.

In order to assist customers in the understanding and validation of requirements, Prospec provide different visual abstractions that present alternative perspectives. For instance, composite propositions depict relations among propositions using symbols, timeline diagrams and, for response patterns only, Petri nets. Prospec generates extended FIL and LTL specifications. An automated transformation from FIL safety formulas to MEDL alarms provides support for run-time verification via the runtime monitor named MaC [KV99]. By automating parts of the formal verification process, Prospec augments the use of formal methods in software development.

2.3. Tracing via FasTLInC

Tracing provides linkages 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.

Unlike traditional traceability models, the tracing approach used in DynaMICs [GM02], 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. Fig. 2 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 [PG96] or SODOS [HW86] to provide this functionality.

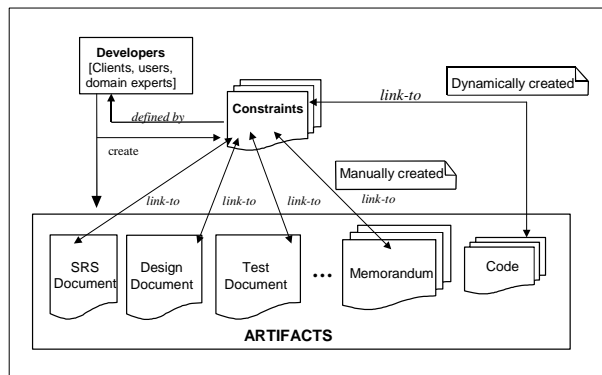


Figure 2: High-level FasTLInC traceability model

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 [GL98].

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. Automated Instrumentation in DynaMICs

3.1. Object-Level Instrumentation

DynaMICs instruments programs at the intermediate- or object-level code [GR01]. One reason for performing the instrumentation at intermediate- or object-code level rather than at the source code level is to avoid the complex analysis required for references with side effects. For example, C and Java programmers idiomatically assign values within conditionals. Suppose that an invariant on variable *c* states that it must be larger than another variable *limit*. Instrumenting the following source code fragment is prone to error since one must realize that checking code must be added within the loop and immediately following the loop.

```
while ((c=fgetc(in) == mode){counter++;}
```

At the object code level, the function call will be identified by its own opcode and the instruction requiring a constraint check will not be lost within a program construct. A similar problem occurs when a function call, method invocation, or creation of a new object instance is passed as an argument in another function or method invocation, as the following Java source code shows:

```
in = new BufferedReader
    (new InputStreamReader (infile));
```

At the Java bytecode level, the mnemonic for each bytecode instruction and the instruction's arguments are examined to determine if a property needs to be checked.

Automated instrumentation can avoid the errors common with manual placement of checks at the source code level. Our case studies have shown that in situations like the previous two examples, while conceptually simple, programmers frequently have difficulty correctly instrumenting code. The problem becomes worse when programs execute concurrent threads as in some architectures. The interleaving of execution of concurrent threads may obscure the need to test constraints in one or both threads.

3.2. Program Analysis and Path Expressions

DynaMICs statically determines the program points at which the properties should be checked [GR01]. This requires program execution flow analysis and the identification of modifications of program variables representing objects or data structures associated with the property. To assist in this effort, a *control flow graph* (CFG) is converted to a *path expression* (PE). One strong point in favor of PE representation is that the identification of execution points that need to be instrumented is simplified. A *basic block* is a sequence of program statements such that control can be transferred only to its first statement and, once this first statement is executed, all the statements in the block are executed sequentially. The basic block and the index of the instruction in the block uniquely identify each instruction in a program. Each basic block contains a unique label. The execution of a program can be seen as a tour among basic blocks.

PEs are regular expressions that provide an efficient approach to path analysis and facilitate processing by automated tools. In a PE, the successor relation is expressed using concatenation. Thus, the PE for a sequence of basic blocks n_1 and n_2 is the string n_1n_2 . Alternate paths, i.e., if-then-else statements, begin with a decision node and are denoted by the symbol "+". Because the high-level programs considered here have a single entry and a single exit, alternate paths always converge to a join node. Iteration in path expressions is denoted by the symbol "*". This indicates that a particular sub-expression may be repeated zero or more times.

Executable Editable Library (EEL) for C and C++ programs [LS95] and JTrek [C01] for Java programs are tools that can be used to create and analyze CFGs. We have implemented algorithms for producing path expressions from CFGs. Restrictions are placed on

programs that can be monitored, in particular programs must be written using structured programming techniques. While other PE algorithms exist, our algorithms take advantage of structured programming style and consider compiler optimizations such as short-circuit Boolean expression evaluation.

Each basic block is annotated with a *write list*. The write list of a basic block is a set of tuples that consist of variables that are modified by instructions in the basic block, operation type, and the instruction index. Each constraint is annotated with a set of *tags*. A tag identifies a location in a program where a property check must be made. It consists of a label of the basic block, the index of the instruction, and an index to the associated property. Each property has an event definition associated with it.

3.3. Immediate Transition Events

For an event definition with an *immediate* transition, PE analysis entails identifying those blocks that are in the same phase as the given event (e.g., input) and that have write lists that contain any variable specified in the variable-set of the constraint.

In the case of an immediate event, the instruction performing a specified action (denoted by the event definition's phase value) upon a variable of interest (VOI) must be tagged so that the check can take place immediately. The algorithms construct all of the event definitions with transition values of *immediate*. Then, every entry in a write list for a basic block is checked to see if any of the instructions performs an operation corresponding to the phase and variable set of the event being examined. If such an operation occurs, then the instruction is tagged. The following two claims are made with respect to the algorithm. Claim 1: Every instruction that performs an operation according to a specified phase on a variable of some event has that instruction tagged. Claim 2: If an entry is tagged, then the variable associated with the event definition of the constraint is modified by this instruction.

3.4. Delayed Transition Events

For analysis of delayed events, it is necessary to instrument at the earliest point in the specified program phase at which it can be guaranteed that no variables from the event's variable-set are updated. For example, for a *delayed-on-input* constraint on variables x and y , the program must be instrumented at the earliest point at which it can be guaranteed that no further read statements for either x or y will occur. The algorithm to tag delayed properties considers that a path expression is a sequence of tokens. As such, the algorithm parses the expression from right to left, searching for the first basic block that modifies a variable of interest, which is in fact the last point in program execution where the variable is

modified. The parser considers the following three cases. The last two require determining if the block of interest occurs within an *if-then-else* or *while* construct. In these cases, the levels of nesting of the constructs in which the block is contained are examined.

Case 1: The last operation occurs in a block that is not contained within an *if-then-else* or *while* body. In this case, the check occurs immediately after the last instruction that modifies any of the elements in the variable-set.

Case 2: If the block is contained within an iterative construct, the check occurs at the beginning of the block immediately outside the while. In a sub-path-expression string such as $[n_1 \cdot n_2]^* n_3 \cdot n_4$, the check would occur before the first statement of block n_3 , which is the block immediately following the outermost “*”.

Case 3: If the block is contained within a selective construct, all paths from the join node of the outermost selective construct to its corresponding conditional node must be evaluated (parsing the PE from right to left). The following cases exist. If the first VOI encountered in the PE is modified within an iterative construct, then proceed as described in Case 2. Otherwise, tag the block associated with the VOI. If a conditional node is encountered during the parsing of a PE and if either both branches have a block that has been tagged or neither branch has been tagged, a recursive call to the algorithm is made to examine the remaining paths in the construct. Otherwise, tag the first block of the branch that does not have a block that has been tagged and make a recursive call to the algorithm to examine the remaining paths. The algorithm terminates when the outermost conditional node is encountered.

The following claims are made with respect to the algorithm. Claim 1: Let P be the set of all tags in a program as constructed by the algorithm. Any event e with a *delayed* transition value is assigned at most one tag p along a path in the PE. Claim 2: For any basic block b that is associated with a tag t , there does not exist a basic block that is a successor of b with an instruction that performs the same phase on the same variables as the event associated with t . Claim 3: If there exists a path where a delayed constraint is assigned a tag, then all paths have a tag associated with the same delayed constraint. The algorithms and associated proofs of correctness are provided in Payne’s work [P03].

5. Summary

The DynaMICs approach supports the specification of properties for runtime software-fault detection and automated instrumentation for a class of non-invariant constraints. In addition, the approach defines a framework for adding linkages between artifacts to support tracing.

Manual instrumentation can be tedious and time-consuming, and therefore increases the potential for human error. Maintenance of the system is also an issue when the system is instrumented manually. There are some tools that automate assertion and constraint checking, but the checks are limited to those that can be defined as invariants, or as pre- and post- conditions. The DynaMICs approach automates instrumentation for a larger class of properties.

Control-flow analysis of the program is vital in order to automate instrumentation of the properties to ensure that they are not misplaced and that all appropriate points of execution are monitored. This requirement dictates consideration of intermediate- or object-code control flow as it is adapted and optimized by the compiler. Instrumentation at a higher level of abstraction cannot control the point at which constraints are checked due to optimizations that occur at compile time. This is true in the case when conditional expressions modify monitored variables by either functional calls or analysis of the result of an assignment or read of a monitored variable. The additional issue for critical constraints, which is not addressed fully here, is the possibility to corrupt code with insertions. To support this process, we describe the algorithms for instrumenting programs. In the end, the case for program instrumentation at the object level is based on the fact that the obvious alternative, i.e., instrumentation at a higher level of abstraction, cannot control the point at which properties are checked.

It is our belief that the techniques described in this paper are applicable to other formal approaches and adaptable to other monitoring tools. For example, the specifications generated by Prospect can be used in model checkers or other runtime monitor systems. In addition, runtime monitoring tools, such as JavaMac, can be extended to include the automated instrumentation approach proposed in this paper.

6. Acknowledgements

This research was partially supported by Consejo Nacional de Ciencia y Tecnología under Contract 68761, NASA project no. NCC5-205, NSF project no. 26-1005-2, and ARL project no. DATM05-02-P-0126.

7. References

- [C01] A. Cohen, JTrek. Formerly available at <http://www.compaq.com/java/download/jtrek/>, 2001.
- [CW96] E. M. Clarke, J. M. Wing, et al., "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, 1996, pp. 626-643.
- [DA98] M.B. Dwyer, G.S. Avrunin, et.al., "Property Specification Patterns for Finite-State Verification," *2nd*

Workshop on formal Methods in Software Practice, Clearwater Beach, Florida, 1998, pp. 7-15.

[GL98] A. Gates and S. Li, "Software Faults and their Detection through DynaMICs," *Proceedings of the IASTED Conference Software Engineering*, 323-327, 1998.

[GM02] A. Gates, and O. Mondragon, "A Constraint-Based Tracing Approach," *Journal of Systems and Software*, Elsevier Science Inc., Amsterdam, vol. 63, 2002, pp. 241-258.

[GR01] A. Gates, S. Roach, et al., "DynaMICs: Comprehensive Support for Run-Time Monitoring," *Proceedings of the Runtime Verification Workshop 2001*, Paris, vol. 55, no. 2, July 2001, pp. 61-77.

[HM96] C. Heitmeyer and D. Mandrioli, eds. "Formal Methods for Real-Time Systems," in *Trends in Software* (no. 5), John Wiley & Sons, New York, 1996.

[H97] G. Holtzmann, "The Spin Model Checker," *IEEE Transactions on Software Engineering*, vol. 23, 1997, pp. 279-295.

[HW86] E. Horowitz, R. Williamson, "SODOS: A Software Documentation Support Environment – Its Use," *IEEE Transactions on Software Engineering*, vol. 12, no. 11, 1076-1087, 1986.

[KV99] M. Kim, M. Viswanathan, et al., "Formally Specified Monitoring of Temporal Properties," in *Proceedings of the European Conference on Real-Time Systems*, 1999.

[LS95] J. Larus, and E. Schnarr, "EEL: Machine-Independent Executable Editing," *ACM SIGPLAN Notices*, vol. 30, no. 6, 1995, pp. 291-300.

[P03] M. Payne, "Automating Instrumentation: Identifying Instrumentation Points for Monitoring Constraints at Runtime," Master's Thesis, The University of Texas at El Paso, Dec. 2003.

[PG96] F. Pinheiro, A. Goguen, "An Object-Oriented Tool for Tracing Requirements," *IEEE Software*, vol. 13, no. 4, 52-64, 1996.

[RJ00] J. Rushby, "Theorem Proving for Verification," in *Modelling and Verification of Parallel Processes* (F. Cassez, ed.), Springer-Verlag, Nantes, France, 2000.

[RT02] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing," *Health, Social, and Economic Research*, Project No. 7007001, Research Triangle Park, NC.