Implementing Rule-based Monitors within a Framework for Continuous Requirements Monitoring

William N. Robinson
Georgia State University
wrobinson@gsu.edu

Abstract

With the increasing complexity of information systems, it is becoming increasingly unclear as to how information system behaviors relate to stated requirements. Although requirements documents and Business Activity Monitoring can provide static and dynamic evidence for requirements compliance, neither provides a formal, real-time presentation of requirements satisfaction.

The REQMON research project is constructing and validating methods and tools for requirements specification and real-time monitoring. The challenge is to simplify monitoring system construction while ensuring the fidelity and expressiveness of its feedback. To address this challenge, our integrative approach leverages a formal monitoring abstraction layer, dynamically configurable distributed monitors, and commercial software to define a theory for specifying, developing, and analyzing requirements monitoring systems.

This article presents an implementation of rule-based monitors, which are derived from system requirements. Such an implementation can simplify the specification of temporal requirements monitors and can be efficient, as our analysis shows.

1 Introduction

Monitoring information systems for requirements compliance is an important and growing problem[15]. Citizens are concerned about the privacy of their information, and the trustworthiness of corporations and government. Consequently, laws have been enacted to ensure policy compliance—from income tax privacy and passport controls, to Sarbanes-Oxley Corporate auditing[22]. The activities of corporations and governments are enacted through their software systems, increasingly so with the trend to digitize processes[16]. Yet, information systems are too complex to determine policy compliance directly, particularly in real-time. The digital trend points to a future society in which there is no visible, or knowable, relationship between information system activities and stated requirements.

Our research on requirements monitoring seeks to increase the visibility of requirements compliance by information systems. Eventually, analysts will use monitoring tools to visualize the extent to which information systems comply with stated requirements. Toward that end, we have defined a requirements monitoring framework, called REQMON.

1.1 Research Objectives

Systematic design of requirements monitoring systems is needed because research has found that systems are often misaligned with their policies[1, 2], there are no systematic design methodologies for requirements monitoring systems, and there is limited support for real-time requirements monitoring[29]. Our monitoring research aims to address these needs through the REQMON framework and supporting tools.

- *Requirements monitoring framework*: The REQMON framework defines: (1) a language for requirements and monitor definitions; and (2) a methodology for defining requirements, identifying potential requirements obstacles, and analyzing monitor feedback.
- *Requirements monitoring tools*: Using widely accepted standards, the REQMON tools provide practical support for development and use of monitoring systems. Analysts apply the requirements analysis tool for requirements definition and analysis; apply the design tool for monitoring systems design and deployment; and view monitored requirements satisfaction with a digital dashboard.

Like an automobile diagnostic tool, the REQMON monitoring tools can be connected to running systems that were not developed with "on board" monitoring instrumentation; web services, dynamic link libraries (DLLs), and byte code insertion enable certain kinds of opportunistic monitoring[3, 17]. Of course, like modern automobiles, information systems can be specifically configured to support requirements monitoring.

1.2 Enterprise Monitoring

Monitoring is particularly helpful for inter-organizational information systems, as typified by governmental agencies and their industrial partners. In support of developing such systems, recent technical research falls into two categories: (1) enterprise modeling, such as Business Process Management (BPM) and Model Driven Architecture (MDA) [12, 24], and (2) distributed architectures and technologies, such as grid computing[11], Enterprise JavaBeans[21], the .NET framework[25], and CORBA[23]. Absent, however, is research into the dynamic monitoring of enterprise systems compliance.

Researchers draw distinctions among policies, goals, and requirements[1, 35]. A *goal* is a desired property of the software and its environment; it describes what "ought" to occur. A *requirement* refines a goal by satisfying three properties: (i) it is described entirely in terms of values monitored by the software; (ii) it constrains only values that are controlled by the software; and (iii) the controlled values are not defined in terms of future monitored values. A *policy* is a goal that is: (i) abstract and broadly scoped, (ii) addresses societal values, and (iii) requires human interpretation; for example, "The system shall protect consumer rights."

Requirement monitoring systems represent the state-of-the-art for monitoring. Theoretical approaches suggest using Linear Temporal Logic (LTL), or similar expressions, for defining requirement monitors; however, only a few research implementations support LTL requirements monitoring. Many commercial-off-the-shelf-systems (COTS) monitor service events. According to Gartner Research, such Business Activity Management (BAM) systems will be one of the top four initiatives driving IT investment and strategy in progressive enterprises[4, 14]—growing from $13.4 billion in 2001 to $22.4 billion in 2005. Although BAM works for narrowly focused applications, like watching for web service failures, monitoring logical relationships among wide-ranging activities is more difficult. In BAM, complex monitor expressions are manually programmed, if they exist at all.

To understand the need for BAM, consider Oregon's groundfish season fishing. August 30, 2004 brought a surprise: Oregon's groundfish season would close in just four days—the Friday prior to the lucrative Labor Day weekend. A Newport, Ore charter company estimated their loss at $100,000 in cancelled trips. Include the other Oregon fishing communities, and their losses of hotel, restaurant, and related services, and the economic impact of the surprise closure is staggering.

Oregon Department of Fish and Wildlife monitors daily fish catches. Ideally, the total fish catch would be updated daily; however, the slips of paper are tallied irregularly. Thus, ODFW was surprised that the quota was nearly met at the end of August 2004, when the total catch was determined. If the actual catch has been monitored daily, then ODFW could have reduced the remaining daily catch limits to salvage a nearly normal season, and millions in fishing related revenues.

To understand the technical descriptions of requirements monitoring, consider privacy requirements monitoring for a Vehicle Miles Traveled (VMT) usage fee system. The efficiencies of modern automobiles have increased the miles travel relative to the fixed fuel state tax; thus, road tax revenues are not keeping pace with usage. A Vehicle Miles Traveled (VMT) tax system is being prototyped by the Road Usage Task Force (www.odot.state.or.us/ruftf) for taxing road usage, as well as region and time of day traveled. An on-vehicle device (OVD), using Global Positioning Satellites (GPS), tracks zone miles traveled—the period, time, and miles traveled within a specified region. Naturally, citizens are very concerned about potential privacy violations of the location information. ODOT seeks to implement the system, while ensuring that the system complies with applicable governmental policies. Moreover, ODOT seeks to gain citizen support by showing that the government-industry collaboration is indeed policy compliant.

To aid acceptance, the following descriptions should be satisfied, ranging from a high-level policy to a low-level obstacle.

```
Policy: The VMT system shall maintain user
privacy
Goal: VMT system shall not maintain location
information
Requirement: The on-vehicle device shall not
store location information
Obstacle: On-vehicle device storage of location
information
```

The potential obstacle, to be monitored, is simply the negation of the requirement, which is derived from the goal. Should an obstructing event occur, the monitoring system needs to raise an alert for the requirement failure, and the associated goal and policy, which are jeopardized.

1.3 Article Overview

Using the VMT usage fee system as an example, this article presents the implementation of monitors within the ReqMon system. Next, a brief summary of related research in presented in section 2. Then, sections 3 and 4 present the ReqMon framework, with special attention given to the monitoring language (§3.2) and its implementation within the Jess rule-based system (§4.1). Finally, the article concludes with a performance evaluation of the monitoring implementation (§5) and a discussion of the framework (§6). To provide context, issues of target system adaptation and monitoring system optimization are raised but undeveloped, so that a more thorough account of the monitor implementation can be achieved.

2 Related Work

A simple two layer conceptual model underlies monitoring systems. The design-time model represents system requirements and other artifacts produced prior to system deployment. The run-time model represents the system execution. A program trace, presented as a (UML) message sequence or (workflow) activity diagram, is a typical run-time model. Monitoring attempts to answer a verification question: does the system—as represented by the run-time model—satisfy the design-time model?

Monitoring systems construct their run-time model by analyzing related events. Figure 1 presents a conceptual illustration of a layered monitoring system architecture. Monitoring systems capture software events, ranging from low-level program methods to high-level business processes. Filtering reduces event tracing and storage to an interesting subset, as a means of improving efficiency. Real-time and long-term analysis of the event repository is presented using a dashboard that shows the health and activities of the target system. Monitoring systems, both commercial and research, have overlapping functionality, but none provide a comprehensive solution.

Monitoring systems do not consider requirements traceability, typically. All too often, Figure 1 represents the entire monitoring system: the design-time model is set aside after system construction. Consequently, analysts must define new monitoring requirements, loosely based on what they know of the original requirements. Moreover, monitoring languages are typically less expressive than requirements languages. Thus, some requirements cannot be monitored. In fact, commercial monitoring systems focus on tracking business processes, represented as activity diagrams. The higher-level business process goals are lost, and thus their satisfaction cannot be determined.

2.1 Monitoring Languages

Providing an expressive language for monitor definition is critical for enterprise systems. At design-time, service requirements often reference message correspondences, temporal relationships, data integrity, and historical information. For example, consider the following abstract requirement that limits the average response time of a service, after another service executes.

```
The average response time of service S1 after
service S2, where (S1.p R S2.p) shall be within
time t.
```

To support analysis of this requirement, both service times, and their properties *p*, must be recorded. To check this requirement, the monitor averages the times between *S1* and *S2* pairs, which satisfy relation *R*; such a relational operator can be used, for example, to check the equality of customer accounts across services.

Enterprise systems include temporal requirements often. As an example, consider the following simple response requirement.

```
If service S1 occurs, then service S2 should
eventually occur (within timeout t)
```

The preceding requirement demonstrates a common temporal pattern. As such, one might expect to find a temporal template in the library of monitor tools, such as Microsoft Operations Manager or Business Activity Monitor (BAM). Instead, commercial tools support monitoring with event based programs, of the following form:

- *When event occurs do something*, where something is a library script, or custom program.

To monitor the preceding requirement, a programmer

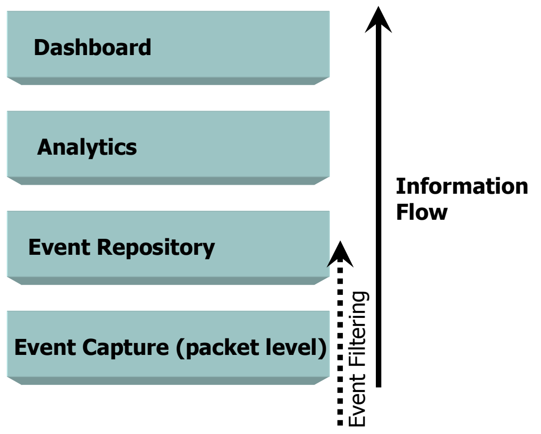

Figure 1 Run-time conceptual architecture.

must translate the implied `S1 eventually leads to S2` to programmatic service accesses. Such requirements are commonly evaluated by through database queries after a polling period elapses. Near real-time monitoring has a long polling period, while real-time monitoring has a short, to zero, polling period. Zero polling can be achieved by explicitly setting timeouts on the associated events. Given the complexity of both temporal logic and real-time programming, such monitor programming is error prone[18]. A more formal approach may reduce monitor specification and programming errors.

Unfortunately, pragmatic barriers reduce the likelihood of formal language usage, including the lack of: good tool support, expertise, good training materials, and process support[32]. A pattern language for formal requirements may facilitate formal requirements usage. Dwyer *et. al.* have analyzed over 500 examples of the kinds of properties that will be used in requirements[6]. They found that nearly all conformed to nine temporal property patterns with five different variable scopes. Although the Dwyer *et. al.* temporal properties are used for program property specifications, they may be applied to monitoring specifications.

3 A Monitoring Framework

Traceability between expressive requirements language and their run-time monitors is central to the REQMON approach. The framework supports this by addressing three interrelated monitoring issues.

- Formalization of high-level goals, requirements, and their monitors
- Automation of monitor generation, deployment, and optimization
- Traceability between high-level descriptions and lower-level run-time events

Exploration of these issues continues in the areas of program verification, requirements engineering, and business process management. REQMON attempts to integrate and extend these disparate efforts.

To provide an overview of the REQMON framework, we present Figure 2, which illustrates the conceptual architecture of REQMON. To use this framework, analysts configure sources and sinks such that monitoring events corresponds to monitoring requirements. An event source is a stream of real-time events, such as provided by instrumented software or network devices. An event sink is a network addressable event destination, which consumes, and potentially persists, events.

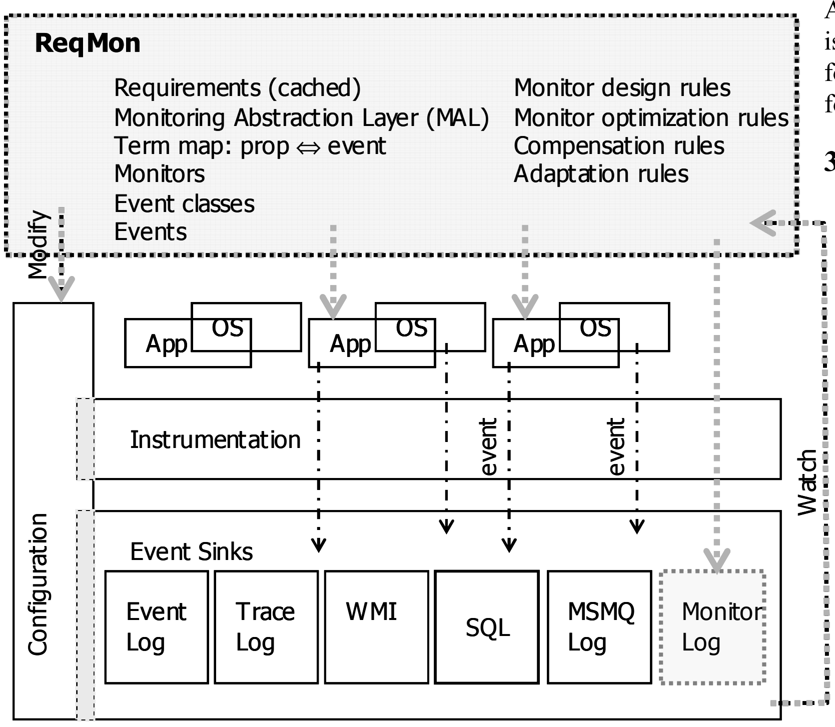

Figure 2 ReqMon conceptual architecture. Shaded elements represent our research extensions to commercial work.

The overall monitoring and adaptation activities proceed as follows:

1. An instrumented target program, outputs trace information.
2. An event transport framework moves trace information and operating system events over a network to a REQMON SERVER.
3. As events arrive, REQMON monitors determine requirements satisfaction.
4. When requirements becomes unsatisfied, adaptation rules execute, updating the target program.

The suite of REQMON tools assist in deriving monitors from requirements and optimizing the monitoring system design.

This monitoring approach integrates requirements language research (KAOS[5], Dwyer *et. al.* patterns[6]) with commercial business process monitoring. Much of REQMON's instrumentation, event transport, and event sinks are provided by commercial tools. For example, event management is implemented using Microsoft Windows Management Instrumentation: (WMI)[33] and Enterprise Instrumentation Framework (EIF)[20]. The resulting system (1) supports real-time logic descriptions, including aggregates, (2) assists deriving monitors from requirements, and (3) associates observed events with monitors, and their associated requirements. It also handles the practical issues associated with (1) distributing monitors, (2) monitoring distributed concurrent business transactions, and (3) instrumenting a variety of systems, such as web services, Seibel, SAP, *etc.* Although commercial systems are used, this is basic research aimed to develop principles for the engineering of requirements monitors for enterprise software systems.

3.1 Methodology

The REQMON approach to monitoring includes three main activities:

1. Define the logical monitoring model
   a. Define goals and requirements
   b. Uncover potential requirements obstacles and derive their monitors
2. Define the monitoring architecture & implementation
   a. Define requirements of the monitoring event sources and sinks
   b. Define a logical-physical mapping to ensure

traceability of events back to requirements

c. Implement and deploy the monitoring system

3. Monitor the enterprise system
   a. Provide high-level feedback on the systems actions and requirements compliance
   b. Execute compensation and adaptation rules when violations occur
   c. If requested, provide high-level feedback on the monitoring system itself, thereby providing historical information used in defining new monitoring optimization rules

The next subsections expand on important aspects of these activities.

3.2 Requirements Language

ReqMon uses the KAOS language, patterns, and some methodology for its requirements, and monitor definitions[5]. The language makes use of real-time temporal operators[19], including: some time in the future (◊), the next state (o), the previous state (●), some time in the past (◆), always in the past (■),always in the future (□), always in the future unless ($\mathcal{W}$), and always in the future until ($\mathcal{U}$). The language also makes use of the standard logical connectives: ∧ (and), ∨ (or), ¬ (not), → (implies), ↔ (equivalent), ⇒ (entails), ⇔ (congruent), with the following relationships.

```
P ⇒ Q iff  □(P→Q)
P ⇔ Q iff  □(P↔Q)
```

Unfortunately, temporal expressions can be difficult to define and understand. Consider the LTL expression of "event *P* occurs between events *Q* and *R*."

```
□ (Q ∧ ¬R → (¬R 𝒲 (P ∧ ¬R)))
```

In an attempt to improve readability and reduce definition errors, Dwyer *et. al.* define nine temporal patterns with five different variable scopes[6]. ReqMon includes the Dwyer patterns and provides requirements templates to aid their usage. Thus, analysts can simply state, "event *P* occurs between events *Q* and *R*" rather than the preceding LTL expression.

According to Dwyer[6], "Scope operators are not present in most specification formalisms (interval logics are an exception). Nevertheless, our experience strongly indicates that most informal requirements are specified as properties of program executions or segments of program executions." For example, property *P* shall be true only after, or between, certain events. Figure 3 illustrates the Dwyer property scopes according to events *Q* and *R*[6], which are supported by ReqMon.

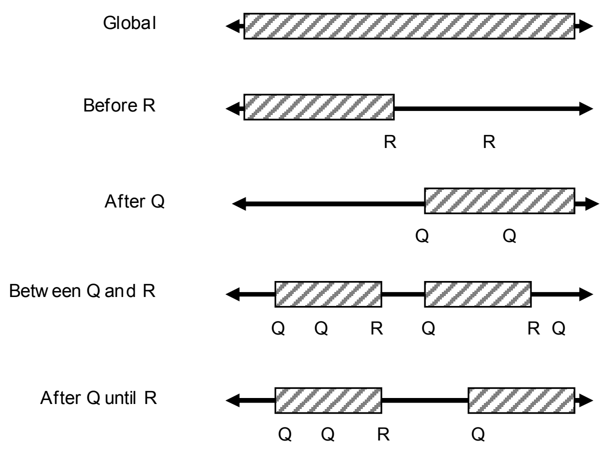

Figure 3. Illustrated times for which properties are valid.

The Dwyer temporal patterns apply within one of the scopes, illustrated in Figure 3; they include universal, absence, existence, bounded existence, response, precedence, chained precedence, and chained response. For example, a universal property is always true, within the scope; whereas, an absence property is never true within the scope. The response pattern, for example, defines properties of the form, `Event` ⇒ ◊ `Response`, where ◊ denotes that the `Response` must eventually (◊) occur after the enabling `Event`.

Monitors watch for requirements satisfaction and violations. Because requirements and their violations and are similar, monitors and requirements share a common language.

Table 1 summarizes some monitor expressions. The language also includes aggregate functions, such as Count, Average, Sum, *etc*. Logical expressions containing these second-order predicates are not analyzed directly; however, they are evaluated appropriately at run-time. As an example, consider the following monitor of average response times.

```
continually, check that before now 10 hours,
average
response(CreditRequest,ConfirmCredit).period < 2
minutes
```

Table 1 REQMON monitoring templates.

`<monitor> :`	`continually | every <period> check that <monitor_spec>`
`<monitor_spec> :`	`<scope> <temporal_patt> | <scope> <agg_function> | <scope> <temporal_patt>.<agg_property> <comparator> <value>`
`<scope> :`	`global | before R | after Q | between Q & R | after Q until R`
`<temporal_patt> :`	`universal | absence | existence | bounded existence | response | precedence | chained precedence | chained response`
`<agg_function> :`	`count | min | max | average | stddev`
`<agg_property> :`	`true | false | period`

As an illustration of defining monitors, reconsider the VMT fee system from the introductory section 1.2. The on-vehicle devices (OVD) are a critical component of the system. Therefore, they include self-diagnostics, which can detect internal malfunctions or tampering. Each OVD must provide periodically its diagnostic report to the monitoring agency, as specified in the following requirement. (Although there is a technical difference between goals and requirements (§1.2), we will use common nomenclature, and simply speak of requirements and requirements monitoring unless a distinction is necessary.)

```
Goal[PeriodicReportSelfDiagnosis]
 UnderResponsibility OVD
 FormalDef
  ∀ o:OVD, r:DiagnosisReport,
    t:Timeout, a:DeviceAuthority
  Raised(ovd,t) ⇒ ◊<t Send(ovd,r,a)
```

This requirement can be violated, as OVDs can fail, due to hardware, software, tampering problems, vehicle non-usage or retirement. Therefore, the average and trend requirements satisfaction are important to know. These are specified in the following requirements.

```
Goal Maximize [ReportSelfDiagnosisConsistently]
  UnderResponsibility OVD
  FormalDef
     ThesholdReportSelfDiagnosis ∧
     TrendReportSelfDiagnosis
Definition[ThesholdReportSelfDiagnosis]
  FormalDef
  Average(PercentSucessful(
    PeriodicReportSelfDiagnosis(*),
    1m),-12m)) ≥ 90%
Definition[TrendReportSelfDiagnosis]
  FormalDef
  Slope(PercentSucessful(
    PeriodicReportSelfDiagnosis(*),
    1m),-6m)) ≥ 0
Definition [PercentSucessful]
  FormalDef
  Satisfied(g,p) / Failed(g,p)
```

The overall goal is improve the reporting of the OVDs. This goal is decomposed into two definitions, which calculate the average success and trend over a sliding time window. Monitors can determine the satisfaction of these goals.

REQMON DESIGNER can derive monitors from definitions conforming to the REQMON templates, such as the Dwyer patterns. Thus, monitors as specified in section 4.1, can be automatically derived from the preceding requirements.

3.3 Event Specifications

Consider monitoring events for the preceding `PeriodicReportSelfDiagnosis` requirement's `Send` predicate:

1. OVD software creates an instance of a class corresponding to the `Send` predicate, as specified in the term map
2. A *watch*, associated with the instrumented program's event source, creates and raises a `Send` event, *e1*
3. Event sinks receive the event, *e1*
4. REQMON receives the event, *e1*, via watching the event sinks.
5. Monitors watch and react to the arrival of REQMON events.

An event specification defines the class of events that a monitor will watch independent of how REQMON receives the event. For the preceding example, the event specification declares a watch for all OVD `Send` events.

Event monitoring configuration requirements specify *how* REQMON receives specified events through certain event sources and sinks. Software events are considered event sources; they can be accessed through a variety of means. For example, web services can be instrumented within the program, the web service protocol stack, web server filters, or a web service gateway; other programs can be instrumented by byte-code insertion[3]. Configuration requirements narrow the event source and sink selection to those that are most appropriate.

As Figure 2 illustrates, REQMON supports sending events to a memory store (WMI log), a file (Event log), a database directly (SQL log), or a store-and-forward queue and then onto a database (MSMQ log)—each subsequent alternative consumes more time, but provides more capabilities.

Sources and sinks can be distributed across a network. For example, a software source on computer *c1*, can send events to a sink, *s2*, residing on another computer, *c2*. Sink *s2* can also serve as a source for sink *s3*, residing on computer, *c3*, which is analyzed by the REQMON SERVER. Thus, events and event analyses are distributable, including over mobile devices. As part of monitoring system design, the REQMON DESIGNER seeks to specify event configuration requirements that optimize monitoring design goals, such as maximal accuracy and minimal overhead and latency.

3.4 Requirements Monitoring

REQMON automatically links raw event data on system usage to measurements on specified requirements. Monitoring determines if specified properties, which "ought" to occur, do in fact occur as desired; it determines requirements satisfaction from observed behaviors[7-10, 26-31]. REQMON has two modes of operation[27].

1. *Model-based monitoring*: REQMON accumulates information that indicates requirements satisfaction or violation.
2. *Statistical-based monitoring*: REQMON accumulates information about reoccurring patterns, using data mining techniques.

REQMON can be configured to raise alerts, or execute a rule-based system, with changing requirements satisfaction.

The REQMON architecture includes a rule-based system, which applies as REQMON receives events.

- *Working memory* is populated with real-time events collected from the targeted system. An SQL database maintains long-term memory, while the rule-based working memory serves as a cache of recent events and facts.
- *Monitors* execute in response to the addition of new real-time events. A monitor is defined by the events to which it responds, and the set of rules that it applies.
- *A vocabulary mapping* defines the translation between the terms of monitored events and the terms in requirements.

3.5 Continuous Monitoring

Analysts are typically concerned with two property evaluations when monitoring the requirements of a running system: (1) satisfaction and (2) violation. In the context of an event monitoring system, consider the requirement, `R1`: "Property *p* shall always hold true." When should the monitoring system report on the satisfaction of `R1`? Technically, its satisfaction cannot be determined until the program has terminated. Yet, analysts want to know the current satisfaction of `R1`, as the system runs. Therefore, continuous monitoring measures satisfaction and violation over the known execution history—for example, the number of violations, and the ratio of success to failure in terms of running time or number of events.

In this continuous monitoring context, reconsider requirement, `R1`, where property *p* is initially true and eventually becomes false: `R1` satisfaction is true and then becomes false. Satisfaction is more interesting where aggregate measures are considered. For example, consider the requirement, `R2`, "The average response time of service *s1* shall be 1 minute." The satisfaction of this requirement can move between true and false throughout system execution. A continuous monitoring system records both of these satisfaction changes as they occur, as well as providing their averages over time frames.

4 Framework Implementation

The REQMON system implements monitors using the Jess rule-based system[13] with a standard SQL database, and event transport is implemented using Microsoft management APIs—specifically, EIF[20] and WMI[33].

4.1 Implementing Monitors

REQMON uses Event Condition Action (ECA) rules, supported by the Jess rule-based system[13]. Its rules have the common form of: `If conditions Then actions`.

A real-time monitor is defined as two rules. The violation rule watches for conditions that imply immediate property violation, while the satisfaction rule asserts success when a property becomes true. To understand why two rules are used, consider the rule-based monitoring of the following event stream, for the property "Switch *s1* shall always be on."

```
1:          (event (switch s1 on))
2:          (event (switch s1 off))
```

The following rules can monitor the switch property.

```
(defrule Switch-Satisfaction
 (event (switch s1 on))
=>
(assert (property Switch satisfied))
(defrule Switch-Violation
 (event (switch s1 off))
=>
(assert (property Switch violated))
```

Neither rule alone can properly monitor the event stream, because rules execute in response to newly asserted facts in real-time. Without the satisfaction rule, the initial success will never be recorded. Similarly, the violation rule is necessary to record the changing status as the switch moves to the `off` position. Both rules are necessary to monitor satisfaction changes in continuous real-time.

The Dwyer temporal patterns can be mapped to Jess rules. Recall that the Dwyer temporal properties include two constructs: scope and pattern. An illustrative template for their Jess rule specification follows, where scope and pattern are mapped to `scope` and `property` clauses, respectively. (In Jess, `?x` represents a variable, `(clause`

`&: (condition))` tests that the clause satisfies the condition.)

```
(defrule <Name>-Satisfaction
 <scope>
 ?facts <- <property pattern>
 (not (property <Name> violated ?facts)
=>
 (assert (property <Name> satisfied ?facts))
(defrule <Name>-Violation
 <scope>
 ?facts <- <negated property pattern>
=>
 (assert (property <Name> violated ?facts))
```

Table 2 shows the mapping of the Dwyer scopes to Jess clauses, which are placed in the `conditions` portion of a rule.

Table 2 Temporal scope for success rules.

Scope	*Jess clauses in scope section*

`Global`	*`Null`*
	`(property ?p)`

`After Q`	`(event ?q&: (eq ?q Q))`
	`(property ?p&: (after ?q ?p))`

`Before R`	`(not (event ?r&: (eq ?r R)))`
	`(property ?p)`

`Between Q & R`
(R must occur)
	`(event ?q&: (eq ?q Q))`
	`(event ?r&: (eq ?r R))`
	`(property ?p&: (after ?q ?p ?r))`

`After Q until R`	`(event ?q&: (eq ?q Q))`
	`(not (event ?r&: (eq ?r R)))`
	`(property ?p&: (after ?q ?p))`

As an illustration of a Dwyer pattern implementation, consider the following three rules, which implement the pattern, "after *s*, *b* responds to *a*". (The other Dwyer patterns, as well as custom patterns, are defined similarly.)

```
(defrule Responds-Satisfaction
 (property ?name Responds enabled)
 (property ?name scope ?pred ?s)
 (property ?name A ?a)
 (property ?name B ?b)
 (event ?eS&: (match ?eS ?s))
 (event ?eA&: (and (match ?eA ?a)
                   (compare ?pred ?eS ?eA)))
 (event ?eB&: (and (match ?eB ?b)
                   (after ?eA ?eB)))
 (not (property
          ?name violated_?eS ?eA $?))
 (property ?name test ?f)
 (test (apply ?f ?eS ?eA ?eB))
=>
 (assert
    (property ?name satisfied ?eS ?eA ?eB))
```

```
(defrule Responds-Violation
 (property ?name Responds enabled)
 (property ?name scope ?pred ?s)
 (property ?name A ?a)
 (property ?name B ?b)
 (event ?eS&: (match ?eS ?s))
 (event ?eA&: (and (match ?eA ?a)
                   (compare ?pred ?eS ?a)))
 (timeout ?name A ?eA)
 (not (event ?eB&: (and (match ?eB ?b)
                        (after ?eA ?eB))))
 (property ?name test ?f)
 (test (apply ?f ?eS ?eA ?eB))
=>
 (assert
    (property ?name violated ?eS ?eA))
(defrule Responds-Violation-Timeout
 (property ?name Responds enabled)
 (property ?name scope ?pred ?s)
 (property ?name A ?a)
 (property ?name timeout ?t)
 (event ?eS&: (match ?eS ?s))
 (event ?eA&: (and (match ?eA ?a)
                   (compare ?pred ?eS ?a)))
=>
 (Timer Start property ?name ?t A ?a))
```

The `Responds-Satisfaction` rule asserts that the property is satisfied, when *b* occurs after *a* within the scope of after *s*. The first four condition clauses retrieve attributes of the property specification. The next three clauses check the occurrence of the events, *s*, *a*, and *b*, as well as their temporal relationships. The *not* clause prevents the rule from completing if there is already a property violation for the events. The test checks relationships constraints among the events.

The `Responds-Violation` rule is similar in form. However, it checks for the occurrence of a timeout between events *a* and *b*. Finally, the `Responds-Violation-Timeout` rule starts a timer after event *a*; it asserts a timeout fact after the specified period.

A `Responds` property monitor is defined by asserting facts, which specify events for monitoring. The following facts define a `Responds` monitor, `X-AB`; the event specifications are elided for brevity—they are XML descriptions embedded with regular expressions.

```
(property X-AB Responds enabled)
(property X-AB scope before(event-spec …))
(property X-AB A (event-spec …))
(property X-AB B (event-spec …))
(property X-AB test same-context)
(property X-AB timeout "1:0")
```

The preceding declarative definition style simplifies monitor specification. For example, a second complete monitor is specified simply by asserting one more fact, such as follows:

```
(property X-AB B (event-spec another spec…))
```

The rule-matching algorithm will create a rule activation for each set of matching facts. Adding a second specification for event *b* will result in two complete rule activations, each matching a given *b* event specification.

Data mining monitors are defined in manner similar to the Dwyer temporal patterns. They analyze trends over historical data. Thus, their application occurs periodically, typically, rather than with individual events. To define a data mining monitor, an SQL database query is first defined. Second, a declarative Jess definition referencing the query is asserted in Jess. Each time the monitor is enabled, its results are posted by the `DB-Query` rule (show below). As an illustration, consider monitoring the success trend for the preceding `X-AB` monitor. The `Percent-X-AB` monitor looks back at the trend over the past 14 days, where percent success is calculated for each day.

```
(property Percent-X-AB DB-Query enabled)
(property Percent-X-AB
                query satisfied X-AB 1d -14d)
(defrule DB-Query
 (property ?name DB-Query enabled)
 (property ?name
      query ?mode ?pname ?period ?timeframe)
=>
 (SQL ?*db* query property
      ?name ?mode ?pname ?period ?timeframe)
```

Our extended Jess rule system, called JessRT, includes functions to interface with a real-time timer (`Timer`), SQL databases (`SQL`), and an event transport system.

4.2 Transporting Events

As REQMON receives events from event sinks, they are asserted into Jess working memory, where monitors react to event arrival. REQMON relies on a management API—specifically, Microsoft EIF[20] and WMI[33], for the specification of event watches, and the transportation of events.

As an illustration, reconsider monitoring OVD's as they `Send` their reports to a common OVD monitoring agency (§3). When each OVD sends a report, a record of the message is added to WMI sink. REQMON, listening to the WMI sink, will then receive this event and assert it in Jess. The follows shows the WMI watch specification for OVD `Send` events.

```
Host:        www.OVDMonitor.com
Namespace:   root\ReqMon\OVD
Query:       SELECT * FROM MonitorEvent
                WHERE Source = 'OVD.Send'
```

4.3 Monitoring the System

As the transported events are asserted into the Jess working memory, the monitor rules execute to update the requirements status.

5 Evaluation

We have implemented monitors using three program architectures: model checking[26], SQL queries[27], and Event Condition Action (ECA) rules. Currently, we use a Jess (ECA) rule implementation.

We analyzed the overhead of the preliminary REQMON monitoring framework. Table 3 presents a summary of the run-time overhead. To compute the overhead, we repeatedly monitored the successful execution of a pair of web services, each of which had a fixed delay of 100 milliseconds to simulate a calculation. The `Responds` monitor was applied in this form: *Web Service 1* $\Rightarrow \Diamond_{\leq t}$ *Web Service 2*. The Jess rule monitor implementation had an overhead of about two percent, while the SQL monitor implementation had an overhead of about 100 percent.

Table 3 shows trace time and monitor execution time. Trace time is all the overhead excepting the time to execute the monitor, either as rules or as program code with SQL queries. Table 3 shows that the Jess monitors are more efficient, and thus tracing occupies a higher percentage of its execution. Conversely, the SQL monitors are less efficient, and thus tracing occupies relatively less of the total time. Database latency accounts for the different in overhead: Jess relies on its in-process database, while the SQL version relies on a connection to an external database. In both cases, the actual requirements satisfaction calculation is relatively slight.

Table 3 Comparison of monitoring overhead.

`Responds`	*SQL Trace*	*SQL Mon*	*Jess Trace*	*Jess Mon*
Percent increase	7.59%	94.94%	0.91%	0.24%

[7]Results are for response monitoring of 1,000 web service requests on a Pentium 4, 3 GHz, 1G memory, Windows XP; the base service time for 1,000 requests with no trace or monitoring is 213.20 milliseconds.

6 Discussion

Many commercial tools support software monitoring. In fact, it is common for Business Activity Monitoring (BAM) tools to provide near real-time reports, such as web service transactions per minute. Typically, they do so by logging transactions to a database cube, whose periodically data mined results are presented. Still, monitoring specific languages are underdeveloped.

Monitoring tools support script execution in response to events. For example, with Microsoft Operations Manager (MOM), an analyst can define a script to run against an event source, such as a BizTalk web service queue. However, to implement REQMON's `Responds` monitor, a script needs to carry out the steps outlined in the ECA rules of section 4.1. It not a question as to whether such languages can implement temporal monitors, but rather how much effort is involved.

Common temporal operators, such as *eventually condition*[34] or *always condition between t1 and t2*[6], must be defined in scripts. In contrast, the REQMON language includes common temporal patterns, which can be extended by analysts. Moreover, the REQMON rule-engine can be embedded within another system, such as MOM.

This article shows how an ECA rule implementation can simplify the specification of temporal monitors. Additionally, our analysis shows that such monitoring can be efficient, at least when compared to a standard database implementation, such as found in BAM systems.

7 References

[1] Annie I. Antón, Julia B. Earp, Colin Potts, and Thomas A. Alspaugh, "The Role of Policy and Privacy Values in Requirements Engineering," presented at IEEE 5th International Symposium on Requirements Engineering (RE'01), Toronto, Canada, pp. 138-145, 2001.

[2] Annie I. Antón, Julia B. Earp, and Angela Reese, "Analyzing Web Site Privacy Requirements Using a Privacy Goal Taxonomy,," presented at 10th Anniversary IEEE Joint Requirements Engineering Conference (RE'02), Essen, Germany, pp. 605-612, 2002.

[3] Geoff Cohen, Jeff Chase, and David Kaminsky, "Automatic Program Transformation with JOIE," presented at Proceedings of the 1998 USENIX Annual Technical Symposium, 1998.

[4] J. Correia and N. Schroder, "BAM: A Composite Market View," Gartner, Inc April 1 2002.

[5] A. Dardenne, Axel van Lamsweerde, and S. Fickas, "Goal-Directed Requirements Acquisition.," *Science of Computing Programming*, vol. 20, pp. 3-50, 1993.

[6] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett, "Patterns in property specifications for finite-state verification," presented at Twenty-First International Conference on Software Engineering, pages, Los Angeles, pp. 411-420, 1999.

[7] M.S. Feather, S. Fickas, Axel van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behavior," presented at Proceedings of the International Workshop on Software Specification and Design (IWSSD'98), Isobe, 1998.

[8] S. Fickas, T. Beauchamp, and A. Razermera Mamy, "Monitoring Requirements: A Case Study," presented at 17th IEEE International Conference Automated Software Engineering, Edinburgh, 2002.

[9] S. Fickas and M.S. Feather, "Requirements Monitoring in Dynamic Environments," presented at Proceedings of the 2nd International Symposium on Requirements Engineering, York, England, pp. 140-147, 1995.

[10] S. Fickas, M. Skorodinsky, and M. Feather, "Sleeping at Night: Building and Monitoring Better Environmental Models," presented at First Workshop on State of the Art in Automated Software Engineering, 2002.

[11] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International J. Supercomputer Applications*, vol. 15, 2001.

[12] David S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*: John Wiley & Sons, 2003.

[13] Ernest Friedman-Hill, *Jess in Action*: Manning Publications Co, 2003.

[14] Mark Hellinger and Scott Fingerhut, "Business Activity Monitoring: EAI Meets Data Warehousing," *Business Integration Journal*, 2002.

[15] Thomas Hoffman, "Sarbanes-Oxley Sparks Forensics Apps Interest," *Computerworld*, vol. 38, pp. 14, 2004.

[16] Ravi Kalakota and Marcia Robinson, *Services Blueprint: Roadmap for Execution*, 1st edition (June 16, 2003) ed: Addison-Wesley Pub Co, 2003.

[17] Han Bok Lee and Benjamin G. Zorn, "BIT: A Tool for Instrumenting Java Bytecodes," presented at The USENIX Symposium on Internet Technologies and Systems, pp. 73-82, 1997.

[18] Kwei-Jay Lin, "Issues on real-time systems programming: language, compiler, and object orientation," in *Advances in real-time systems*: Publisher Prentice-Hall, Inc., 1995, pp. 335 - 351.

[19] Z. Manna and A. Prueli, *The Temporal Logic of Reactive and Concurrent Systems*: Springer-Verlag, 1992.

[20] Microsoft, "Enterprise Instrumentation Framework (EIF)," Microsoft Corp. http://msdn.microsoft.com/vstudio/productinfo/enterprise/eif/ 2003.

[21] Sun Microsystems, "Enterprise JavaBeans Specification, Version 2.0," Sun Microsystems 2001.

[22] John Montana, "Sarbanes-Oxley one year later," *Information Management Journal*, vol. 37, pp. 18, 2003.

[23] OMG, "CORBA Components: Joint Revised Submission," Object Management Group August 1999.

[24] OMG, "OMG Model Driven Architecture," 2003.

[25] D.S. Platt, *"Introducing Microsoft .NET*, Second Edition ed: Microsoft Press, 2002.

[26] W.N. Robinson, "Monitoring Software Requirements using Instrumented Code," presented at IEEE Proceedings of The 35th Annual Hawaii International Conference on Systems Sciences, Hawaii, 2002.

[27] W.N. Robinson, "Monitoring Web Service Requirements," presented at 11 IEEE International Conference on Requirements Engineering, Monterey Bay, CA, pp. 65-74, 2003.

[28] W.N. Robinson, "Monitoring Requirements Development with a Goal Model, in," in *The Method Engineering Textbook*, M. Jeusfeld, M. Jarke, and J. Mylopoulos, Eds.: MIT Press, 2004.

[29] W.N. Robinson, S. Pawlowski, and V. Volkov, "Requirements Interaction Management," *ACM Computing Surveys (CSUR)*, vol. 35, pp. 132 - 190, 2003.

[30] William N. Robinson, "Monitoring Web Service Interactions," presented at Workshop on Requirements Engineering in Open Systems (REOS'03), In conjunction with 11th IEEE International Requirements Engineering Conference (RE03). Monterey Bay, CA, pp. 12-14, 2003.

[31] William Robinson, N. and Suzanne Pawlowski, D., "Managing requirements inconsistency with development goal monitors," in *IEEE Transactions on Software Engineering*, vol. 25, 1999, pp. 816-835.

[32] D.S. Rosenblum, "Formal Methods and Testing: Why the State-of-the-Art Is Not the State-of-the-Practice," *(ISSTA '96/FMSP '96 Panel Summary), ACM SIGSOFT Software Engineering Notes,*, vol. 21, pp. 64-66, 1996.

[33] Craig Tunstall and Gwyn Cole, *Developing WMI Solutions: A Guide to Windows Management Instrumentation*: Publisher: Addison Wesley Professional, 2003.

[34] A. van Lamsweerde, R. Darimont, and P. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt," presented at IEEE, Second International Symposium on Requirements Engineering, pp. pp. 194-203, 1995.

[35] Axel van Lamsweerde and Emmanuel Letier, "Handling obstacles in goal-oriented requirements engineering," in *IEEE Transactions on Software Engineering*, vol. 26, 2000, pp. 978-1005.