

Monitoring Software Security Requirements using Instrumented Code

William N. Robinson
Department of Computer Information Systems
Georgia State University
Atlanta, GA 30301-4015 USA
wrobinson@gsu.edu <http://cis.gsu.edu/~wrobinso>

ABSTRACT

Ideally, software is derived from requirements whose properties have been established as good. However, it is difficult to define and analyze requirements. Moreover, derivation of software from requirements is error prone. Finally, the installation and use of complied software can introduce errors. Thus, it can be difficult to provide assurances about the state of a software's execution.

We present a framework to monitor requirements of software as it executes. The framework is general, and allows for automated support. In this paper, we introduced the framework, and show how Java code can be instrumented and monitored by a model checker. We illustrate our current automated support using the widely known problem of the Dining Philosophers. From this exemplar, we suggest how the approach may be applied to address security concerns such as those that arise during e-commerce transactions.

1. INTRODUCTION

Software is pervasive. So too are software bugs. Such glitches in the execution of software can cause millions of dollars in lost revenue, as evidenced by recent denial of service attacks[9]. Worse still, lives can be lost, as documented in the case of the Therac-25 computer controlled radiation therapy machine[14].

Producing error free software is difficult. Two complimentary approaches addressing this problem are requirements analysis and software testing. In requirements analysis, the descriptions of the software "to be" is carefully scrutinized. Once the requirements meet certain properties, such as consistency and completeness, software is developed according to the requirements specification. Further down the software life-cycle, software-testing checks that the derived software satisfies the requirements specification. In certain cases, developers must rely more on software testing or run-time requirements monitoring, as a complete analysis of requirements can be intractable.

Consider the CCITT X.509 (1989) protocol for signed secure communication between two parties[12]. It can be used to establish

session keys that allow for authenticated and confidential communication. A goal of the protocol is to prevent the occurrence of an agent, A, sending a message to an agent C, while intending to send the message to agent B. However, proving that this goal is always met is difficult.

The protocol has only three messages:

Message 1 A ? B : A, {T_a, N_a, B, X_a, {Y_a}K_b}K_a⁻¹

Message 2 B ? A : B, {T_b, N_b, A, N_a, X_b, {Y_b}K_a}K_b⁻¹

Message 3 A ? B : A, {N_b}

Above, A and B are the communicating agents; T_j are timestamps; N_j are nonces (a unique random number); X_i and Y_i and user data; and K_i and K_i⁻¹ are public and private keys, respectively. An agent can apply B's public key K_b to data Y_a, {Y_a}K_b, to encrypt the data according to that key. Applying A's private key, K_a⁻¹, to a message, produces a digital signature of A that can be checked by anyone who uses A's public key, K_a.

The CCITT X.509 protocol appears secure enough; however, it is susceptible to two kinds of attack. Below, a sketch of these attacks is presented to provided a general understanding of the protocol's complexity. This will be used show the difficulty in model checking the protocol.

(1) *Reuse of confidential data by imposter.* An imposter, C, can intercept message 1, strip off the signature, K_a⁻¹, and send the message onto B with its own signature.

Imposter Message 2 C ? B : C, {T_c, N_c, B, X_c, {Y_a}K_b}K_c⁻¹

(2) *Replay of message to gain authentication.* An imposter, C, can intercept messages between agents A and B so as to make B believe it is communicating with A, when it is really communicating with C. C begins the attack by capturing and resending an old message sent from A to B. The message is then sent from C to B.

Imposter Message 1 C ? B : A, {T_a, N_a, B, X_a, {Y_a}K_b}K_a⁻¹

If the time is sufficiently close to T_a, then B will not notice the reuse of the message. Specifically, the reused nonce, N_a, will not be checked, so the reuse goes undetected. Next, B will reply with a new message intended for A.

Imposter Message 2 B ? C : B, {T_b, N_b, A, N_a, X_b, {Y_b}K_a}K_b⁻¹

Next, C can begin authentication with A. In reply to A's new

message, C will reply with B's nonce, N_b , of message 2. Finally, A will reply to C using the signature needed for C to send a new message to B.

Here, we are not concerned with the details of these attacks. Rather, we are concerned with how they might be detected.

In general, to detect the malicious manipulation of a communication protocol, one can simulate an agent that executes the protocol with malicious intent. Then, one can check for conditions, such as "Can an agent, A, send a message to agent C, while intending to send the message to agent B?" However, checking the effect of a malicious agent on a protocol can be intractable. For example, Josang estimates that checking the CCITT X.509 protocol for the Replay Attack will involve approximately 10^{19} states[13]. Thus, automated model checking is not practical. However, monitoring is a practical solution.

2. REQUIREMENTS MONITORING OF RUNNING SOFTWARE

Requirements analysis and software testing typically ends after the software is deployed. If testing does continue, it is done "at the factory", rather than at the customer's installation site. Yet, many software errors arise due to changing or unforeseen circumstances found at the installation site. *Perpetual testing* aims to address this gap.

Perpetual testing treats testing as a set of activities continually aimed to improve quality through different generations of a product. Currently, the perpetual testing research is addressing issues closer to software design than the high-level software requirements. For example, Butkevich *et. al.*, describes an extension to Java that enables static and dynamic checking of object interaction protocols (i.e., sequence diagrams) as part of a Java program[24].

Researchers in *requirements monitoring* aim to fill the gap between requirements analysis and perpetual testing. In requirements monitoring, high-level software, or systems requirements, is the focus of study. The goal is to continually analyze requirements, from development through deployment to eventual system retirement. An important requirements monitoring activity concerns the run-time monitoring of requirements.

2.1 Execution Monitoring of Requirements

Execution monitoring of requirements is a technique that tracks the run-time behavior of a system and notes when it deviates from its design-time specification. Requirements monitoring is useful when it is too difficult (e.g., intractable) to prove system properties. To aid analysis, assumptions are made as part of the requirements definition activity. The requirements and assumptions are monitored at run-time. Should any such conditions fail, a procedure can be invoked (e.g., notification to the designer). Note, such

monitoring is different from exception handling in that it: (1) considers the combined behavior of events occurring in multiple threads or processes over time, (2) links run-time behavior with the actual design-time requirements, and (3) provides (potentially) sufficient information to allow for the run-time reconfiguration of software or software components.

Fickas and Feather proposed requirements monitoring to track the achievement of requirements during system execution as part of an architecture to allow the dynamic reconfiguration of component software[8]. Feather has produced a working system, called FLEA, that allows one to monitor events defined in a requirements monitoring language[6][7]. FLEA captures interesting events as assertions in a database. When a monitored condition occurs, its defined action is executed. Thus, monitoring mainly consists of the translation of requirements monitoring descriptions to database triggered actions.

Fickas and Feather illustrate the execution monitoring of requirements in the context of monitoring the requirements of a software license server. When the license server fails to satisfy its requirements (e.g., a user shall be granted a license in 90% of their requests) due to a change in the system environment, the system notifies an administrator. Emmerich *et. al.* (and others [21]) have since illustrated how the technique may be used to monitor process compliance[4]; for example, organizational compliance to ISO 9000 or IEEE process descriptions[20].

Girgensohn *et. al.* created *expectation agents* to monitor the actual use of a system[5]. Developers define software user expectations, such as, "validate the customer address before configuring the customer's services". Then, agents monitor the system's use. When use of the system does not match the defined expectations, "agents may perform the following actions: (1) notify developers of the discrepancy; (2) provide users with an explanation based on developers' rationale, (3) solicit a response to or comment about the expectation." [5] In this manner, expectation agents monitor the satisfaction of developer requirements (expectations) during the system.

2.2 Article Overview

This article describes research that builds on prior work in requirements monitoring. A goal of this research is to continually analyze requirements during the run-time execution of software. Next (§ 3), we present our overall requirements monitoring framework for software execution. An important contribution of this work concerns the continual model checking of running software. This is illustrated with an example in section 4. If such requirements monitoring can be effectively applied in practice, then it can provide important assurances for business communications, such as on-line e-commerce transactions. This topic is explored in the concluding

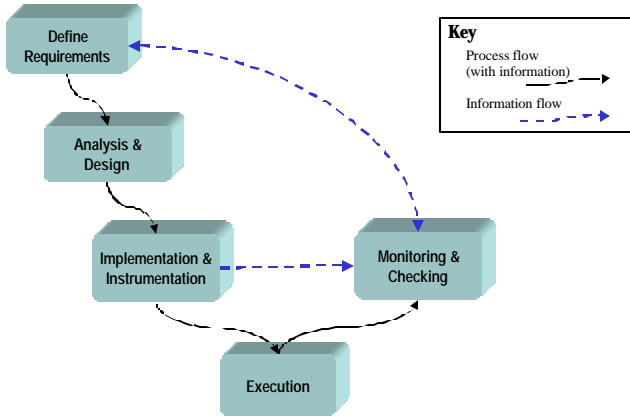


Figure 1. An illustration of a requirements monitor for executing software (RMES) framework.

section 6. There, this article concludes model checking can provide substantial support for monitoring requirements during the run-time execution of software.

3. A REQUIREMENTS MONITORING FRAMEWORK

Figure 1 illustrates our requirements monitor for executing software (RMES) framework. It is essentially a refinement of the model described in [6]. Each of the major activities are described in the following sections.

3.1 High-level Requirements

The framework relies on the description of high-level software requirements. These requirements may include natural language text; however, the framework assumes some formal definitions of desired predicates over objects and relationships. More specifically, requirements that refer to conditions on states are most easily analyzed. For example, the KAOS language can be used for this purpose[3].¹ The following ForkRequestSatisfied achieve goal illustrates one such KAOS requirement. The goal simply indicates that a philosopher, implemented as a software process, should eventually have access to a (shared) fork. (Section 4 presents this example problem in more detail.)

```
Goal Achieve [ForkRequestSatisfied]
InformalDef
  "If a philosopher requests a fork, then
  eventually it will be using the fork."
FormalDef
  ? p:Philosopher, f:Fork
  Requesting(p,f) ? ? Using(p,f)
```

Monitoring the satisfaction of such high-level requirements is the

¹ KAOS formal definitions use temporal logic operators[19]. Here, 0 means in the next state. Other operators are included for the previous state (*), some time in the future (?), some time in the past (≪), always in the future (≧), always in the past (≦).

purpose of the framework.

3.2 Analysis and Design

The framework assumes that the high-level software requirements will be translated into a more development-oriented analysis and design model. For example, the Unified Modeling Language (UML) can be used as the modeling language. This intermediate (UML) model simplifies traceability maintenance between the high-level requirements and the software. Such linkage allows the execution monitor to describe software failures in terms of requirements or the UML model. Since developers are familiar with UML, and it can have a direct correspondence to the software, we have found this intermediate representation useful. However, as long as one can maintain traceability between code and requirements, it is not necessary to have the intermediate UML model.

3.3 Implementing and Instrumenting

The framework assumes many of the high-level requirements will be implemented in software. Moreover, there must be *static traceability* between part of the software and the requirements. For example, a KAOS agent definition, such as the following Philosopher, can be traced to its Java class definition.

```
// KOAS definition...
Agent Philosopher
Has
  left_fork : Fork
  right_fork: Fork
  status : {ready, running, suspended,
  dead}

// Java class definition (manually) derived
KAOS definition...
class Philosopher extends Thread {
  private Fork left;
  private Fork right;
```

In addition to the static traceability of definitions, there must be *dynamic traceability* of class instances. When the software executes, the monitor must be able to distinguish the different instances of the defined classes. For example, the monitor must be able to distinguish the actions of two instances of the same agent type (e.g., knowing that Philosopher1 actions are different from Philosopher2 actions).

Instrumentation can enable a monitor to track the activities of relevant objects. Instrumentation is the insertion of informative statements into software for the purpose of monitoring. Later, when the instrumented software executes, the informative statements provide a stream of information which can be interpreted by the monitor.

Typically, the source code of software is instrumented. However, it is also possible to instrument compiled code (e.g., [11]). For example, our implementation of the RMES framework uses Joie to instrument Java class files (compiled Java files)[1].

Instrumenting compiled code does provide for monitoring of

software without source code. However, there is still the problem of interpreting the monitored stream. For example, given the monitored stream of actions concerning the compiled Java classes, methods, and variables, one must map those names back to the software requirements in order to interpret the actions.

3.4 Monitoring

The monitor must continually view the stream of (instrumented) activities and interpret their meaning. It is important that the monitor notice when the software: 1) has violated a requirement, or 2) is about to violate a requirement, or 3) may be about to violate a requirement. However, this can be difficult given the tiny stream of information it may receive. For example, the following stream is output from an instrumented dining philosophers compiled program.

```
Philosopher[0].eat()V
Fork[0].take()V
Philosopher[1].eat()V
Fork[1].take()V
Philosopher[2].eat()V
Fork[2].take()V
Philosopher[3].eat()V
Fork[3].take()V
```

Each line is of the following format: *classname[instance number].method(parameters)returnType*. No doubt, it's not clear (yet) that the above monitor stream shows a deadlock of four dining philosophers.

To interpret the monitor stream, the monitor can use a model. In our implementation of the RMES framework, a formal automata-based model is checked using a model checker. The formal model is generated automatically from the Java source code.

3.4.1 Model Checking

Model checking is an operational exploration of state-based models. Such analyses can prove that specified logic conditions will, or will not, occur in a modeled state of a system satisfying the requirements.

Model checking of requirements is typically applied as follows:

1) a portion of the requirements specification is translated into a formal automata-based model, 2) important requirements properties, such as liveness, are defined as logical properties of the model, 3) a model checker (e.g., Spin[10]) is used to exhaustively check all states of the model for violations of the specified properties.

In our implementation of the RMES framework, model checking is used as follows:

- (3) Requirements are defined.
- (4) A UML model design is derived from the requirements.
- (5) A Java program is derived the UML model.
- (6) The compiled Java classes are instrumented.
- (7) Periodically, a copy of the Java source is modified to reflect

the activities observed in the monitor stream.

- (8) A Promela model is semi-automatically derived from the modified Java source code.
- (9) The modified Java source code is model checked, thereby checking the actions of the running software.

To clarify, steps 5 - 7 are further elaborated next. After the specified period (in 5), the monitor checks the complete (growing) monitor stream. It does so as follows:

- (1) The monitor stream is translated back into Java statements. These statements control the sequence of method calls in the original Java program. Thus, this "streamlined" Java source code has no loops or conditionals. It simply represents the sequence of method calls that have occurred. In the case of multi-threading, it is necessary to define such a sequence for each thread.
- (2) A Promela model is semi-automatically derived from the modified Java source code. A program, java2spin does this translation[2]. (It does not handle all Java program constructs. When it can, this process is fully automated.)
- (3) Requirements, including "avoid near failures", are translated into the linear temporal logic of Spin.
- (4) The Promela model is checked against the requirements using the Spin model checker. Failures, indicated by Spin, mean that the stream of actions performed by the program did not satisfy the requirements.
- (5) If a failure occurs, the monitor sends a notification detailing the current state of the software and how it has failed a requirement.

3.4.2 Checking for Near Failures

At first, it may appear that the above procedure simply sends a notification when a high-level requirement fails. However, the clarification step 3 allows for the definition of "avoid near failures" requirements. For example, in order to be notified of a near failure, one might add the following "avoid near failure" requirement to the dining philosophers system.

```
Goal Avoid[WaitingOn2Forks]
InformalDef
  "Do not allow more than two philosophers
  to be waiting on a fork request. (In the case
  of four philosophers.)"
FormalDef
  ? p1, p2 :Philosopher, f1, f2 :Fork
  ! ? (WaitingOn(p1,f1) ?
      WaitingOn(p2,f2))
```

If the monitor sends a notification that the above `WaitingOn2Forks` requirement fails, then there may be time to prevent the system from deadlocking, and thus violating a high-level requirement. Of course, this and many similar "avoid near failure" requirements can be made part of the high-level requirements. However, considering all cases, even if possible, can make the requirements, requirements analysis, and software unwieldy, if not intractable—as illustrated with the CCITT X.509 protocol.

4. AN ILLUSTRATIVE EXAMPLE

The widely known problem of the dining philosophers will serve as an exemplar for our approach to RMES framework. This prob-

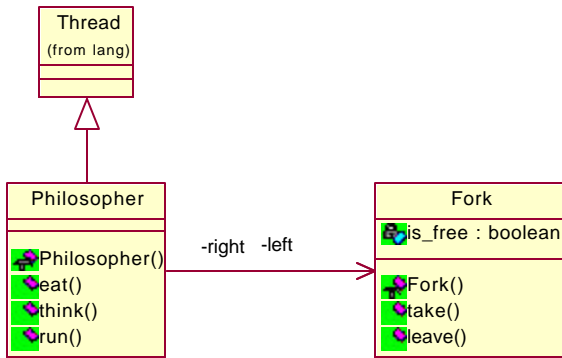


Figure 2. UML class diagram representing the Philosopher and Fork classes.

lem is especially useful, as it demonstrates troubles that can occur among multiple processes with shared resources. Such interactions can occur with today’s on-line e-commerce transactions.

Consider four philosophers sitting around a table. The philosophers can either eat or think. Thinking does not require a resource, whereas eating requires two forks. There are only four forks; each is between a pair of philosophers. A philosopher can only share the use of forks that are immediately to his left or right. The problem is to guarantee mutually exclusive access to the forks, while preventing deadlock or starvation. A deadlock can occur when all philosophers lift up their left fork, and then deadlock as they try to lift up the right fork that is already taken.

4.1 High-level Requirements

The main high-level requirement we will focus on is the `ForkRequestSatisfied` of section 3.1. In combination with our design, it will ensure that a Philosopher will not starve, nor will it become deadlocked.

4.2 Analysis and Design

Figure 2 shows a UML Java design model of the dining philosophers. Figure 3 shows UML collaboration diagram illustrating how the four instances of philosophers and forks are related. (The lines are links that indicate communication among the objects.) Note that this design (and resulting Java program) specifies the number of philosophers and forks *a priori*.

Note, it is possible to directly model check the UML design model. For example, the vUML tool can derive a Spin model and present failures in terms of UML[16]. However, such analysis is only available for smaller models (like the dining philosophers). Larger models, entailing larger state-spaces, cannot be sufficiently explored by model checkers (e.g., [17]). In contrast, the model checking of one specific run of a program is very efficient.

Model-checking a program trace is still a complex process. Each program state must be checked for compliance with linear temporal logic (LTL) formulas[19]. An LTL formula is more

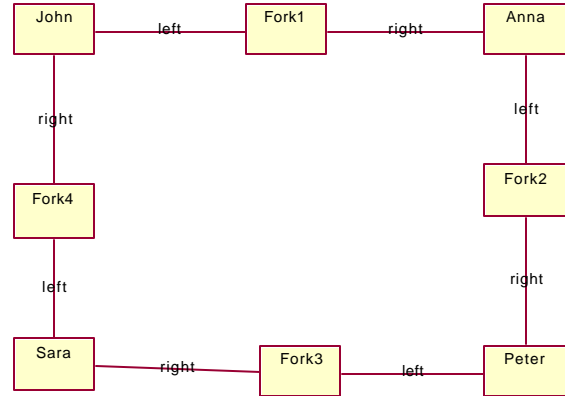


Figure 3. UML collaboration diagram representing the connections among Philosopher and Fork objects.

abstract than program state; it may reference abstract and arbitrary program states, as well as various times (past, present, future). Yet, a model checker can efficiently analyze any properly instrumented program.

Rather than model checking, one might add program assertions to provide assurances about a running program’s behavior[18]. Assertions can be complex and abstract. Program state information can be tracked by specialized “assertions”, thereby allowing LTL references. In fact, one can custom build an LTL model checker for a specific program’s execution using assertions. However, it is easier, more reliable, and more general to use a model-checking tool for that purpose. Rather than insert various assertions and state tracking functions, one can simply express LTL’s and then automatically instrument and check the program.

4.3 Implementing and Instrumenting

The Java class definitions for the philosopher and fork are shown in figure 4. The main program that executes, includes the following statements to initialize the program.

```

phil1 = new Philosopher(fork1, fork2);
phil2 = new Philosopher(fork2, fork3);
phil3 = new Philosopher(fork3, fork4);
phil4 = new Philosopher(fork4, fork1);
phil1.start();
phil2.start();
phil3.start();
phil4.start();
  
```

After the Philosopher and Fork classes were compiled, they were instrumented using Joie. Figure 5 shows the Java class information what was added to the Philosopher and Fork classes. Notice that, as part of the mixin, an initialization method was added. This `init()` method tracks the specific instances of a class via a unique object number. This number is referenced in the `traceMsg` method. Thus, the values appear in the output of section 3.4.

4.4 Monitoring

To monitor, the monitor program continually checks the monitor stream. For this simple example, the monitor stream is simply text output to standard out. (See the `traceMsg` method of figure 6.) However, the output could as easily be sent to an on-line database

```
class Philosopher extends Thread {
    private Fork left;
    private Fork right;
    Philosopher(Fork f1, Fork f2)
    {
        left = f1;
        right = f2;
    }
    public void eat()
    {
        left.take();
        right.take();
    }
    public void think()
    {
        left.leave();
        right.leave();
    }
    public void run()
    {
        while (true) {
            eat();
            think();
        }
    }
}

class Fork
{
    private boolean is_free;
    Fork()
    {
        is_free = true;
    }

    public synchronized void take()
    {
        while (!is_free)
            try {
                wait();
            } catch (InterruptedException e)
            {return;}
        is_free = false;
    }
    public synchronized void leave()
    {
        is_free = true;
        notify();
    }
}
```

Figure 4. The Philosopher and Fork Java class definition.

```
public class TraceMixin {
    // The current count of object (instances)
    // that are being traced.
    private static int OIDcount = 0;
    // The object's ID.
    private int OID;

    public void init () {
        setCurrentOIDcount();
        incrementOIDcount();
    }
    // More definitions (elided)...
    public void traceMsg(String s) {
        System.out.println(this.getClass().get-
        Name() + "[" + OID + "]" + s);
    }
}
```

Figure 5. The Java definition of the TraceMixin class. These definitions were added to each class in the compiled code.

```
class Ph1 extends Philosopher
{
    Ph1(Fork f1, Fork f2)
    {
        super(f1,f2);
    }
    public void run()
    {
        eat();
        // Other steps follow here...
    }
}
```

Figure 6. An updated Java philosopher definition for philosopher 1.

where database triggers would monitor the stream.

4.4.1 Model Checking

The monitor continually checks the output by creating a modified Java program and then checking it in Spin. To modify the program, a program (`monitor2Java`) generates updated Java definitions from the monitor stream. Figure 6 shows an updated definition for instance number one of the Philosopher class. Note, it is necessary to define subclasses (`extends`) of Philosopher because we need to define the prior monitored actions of an instance of the Philosopher class. The other subclasses of Philosopher (`Ph1` to `Ph4`) are similarly defined.

To check the requirements on the program execution, the `java2spin` program is used to create a Promela definition of the modified Java program. Finally, this model (called `dp.spin`) is checked. To do so, each high-level requirement or “avoid near failure” requirement must be added to the model using Spin. However, Spin cannot under requirements such as `ForkRequestSatisfied` of section 3.1. Instead, these requirements must be manually mapped to definitions in the underlying model as created by

java2spin. (This can be difficult, as the java2spin model can be non-intuitive.)

Consider translating the `Avoid[WaitingOn2Forks]` “avoid near failure” requirement. This first must be specified in terms of the Spin definitions. In the example of our `dp.spin`, the following terms are used.

```
#define waitF1(sync_def_Fork[1].nwait > 0)
#define waitF2(sync_def_Fork[2].nwait > 0)
#define waitF3(sync_def_Fork[3].nwait > 0)
#define waitF4(sync_def_Fork[4].nwait > 0)
#define waitOn2((waitF1 && waitF2) ||
(waitF2 && waitF3) \
  || (waitF3 && waitF4)
  || (waitF4 && waitF1))
```

Once so defined, Spin can test for the occurrence of `<>waitOn2`.

Finally, as a result, Spin can detect the violation of requirements such as `Acheive[ForkRequestSatisfied]` and `Avoid[WaitingOn2Forks]` through monitoring of the monitor stream output. For example, after monitoring the output shown in section 3.4, Spin detects that `Acheive[ForkRequestSatisfied]` fails. That is, a deadlock occurred when all philosophers acquired their left fork, and then “blocked” waiting as they tried to acquire their right fork that was already taken.

5. FUTURE RESEARCH

There are a number of directions this research is proceeding. They include the following.

- Improve code to model checker translation. Java2Spin does not handle all Java constructs (e.g., polymorphism, exceptions). Also, Spin does not support many programming related constructs (e.g., pointers, dynamic memory allocation/deletion).
- Improve translations to/from requirements and the model checker. Currently, these translations are manual. We would like to provide some automated assistance.
- Improve the explanations. Spin can provide a trace that led up to the requirement failure; however, these explanations must be translated from the Spin mode back to the design or requirements model.
- Facilitate the evolution of a running system. Eventually, we would like to provide sufficient feedback to guide automate program modification as a means to overcome systematic requirement failures.

The above topics will improve the requirements monitoring tool. As that tool becomes more usable, it will be applied to case studies. Future studies will include e-commerce protocols, such as the CCITT X.509 signed secure communication protocol, and the secure transaction protocol (STP)[25].

6. CONCLUSIONS

We have presented a requirements monitoring framework for the monitoring of executing software. We have described automated tool support that simplifies the application of the framework. In

doing so, we have described how model checking can provide substantial support for requirements monitoring.

The CCITT X.509 signed secure communication protocol consists of only three messages. Yet, it illustrates how difficult it can be to prove that one has discovered all scenarios for a successful attack. Requirements monitoring is a practical alternative to complete *a priori* requirements analysis. In particular, requirements monitoring via model checking is efficient, sound, and safe means to provide assurances on run-time behavior.

CCITT X.509 authentication messages can be monitored as illustrated with the dining philosophers problem. In the case of X.509 authentication, monitoring requires knowledge of multiple agents and their message history (nonce usage). Tracking this within agents via assertions poorly mixes individual agent protocol function with general protocol security—a non-cohesive design. Instead, the behaviors can be monitored in a separate “security agent” that relies on the sound and general reasoning of a model checker.

Requirements monitoring will become a necessity as more relationships are managed through on-line transactions. In the past, much effort was placed on ensuring that a transaction completed properly. Thus, the two-phase commit protocol is common place in databases. However, more elements of transactions are being distributed across the internet. This has led to protocols to ensure secure transactions, such as STP[25]. In the future, more of us may be depending on agents that employ complex negotiation protocols on our behalf[22][23]. Hence, we may also be depending on security agents to ensure appropriate behavior.

7. REFERENCES

- [1] Geoff Cohen, Jeff Chase, and David Kaminsky, Automatic Program Transformation with JOIE by in Proceedings of the 1998 USENIX Annual Technical Symposium.
- [2] C. Demartini, R. Iosif, R. Sisto A Deadlock Detection Tool for Concurrent Java Programs Software - Practice and Experience, Vol. 29, No. 7, July 1999, pp. 577-603, Wiley
- [3] Dardenne, A., van Lamsweerde, A., Fickas, S., Goal-Directed Requirements Acquisition, *Science of Computer Programing*, 20 1993, 3-50.
- [4] Emmerich, W., Finkelstein, A., Montangero, C. & Stevens, R. "Standards Compliant Software Development" in *Proc. International Conference on Software Engineering Workshop on Living with Inconsistency*, (IEEE CS Press), 1997
- [5] A. Girgensohn, D. Redmiles, and F. Shipman, Agent-Based Support for Communication between Developers and Users in Software Design. In *Proceedings of the 9th Knowledge-Based Software Engineering (KBSE-94) Conference*, Monterey, CA: IEEE Computer Society Press, pp. 22-29, 1994.
- [6] Feather, M.S., Fickas, S., van Lamsweerde, A., Ponsard, C., Reconciling System Requirements and Runtime Behavior, *Proceedings of the International Workshop on Software Specification and Design (IWSSD'98)*, Isobe, IEEE CS Press, April, 1998.
- [7] Feather, M.S., FLEA : Formal Language for Expressing Assumptions Language Description, June 25, 1997.
- [8] Fickas, S., Feather, M.S., Requirements Monitoring in

- Dynamic Environments, *Proceedings of the 2nd International Symposium on Requirements Engineering*, IEEE Computer Society Press, York, England (March 1995) 140-147.
- [9] Ann Harrison and Kathleen Ohlson, Surviving Costly Web Strikes, *ComputerWorld*, IDG.com, February 21, 2000.
 - [10] Holzmann, G.J., The Model Checker Spin, IEEE, *Transactions on Software Engineering*, 23, 1997, pp. 279-295.
 - [11] Han Bok Lee and Benjamin G. Zorn. BIT: A Tool for Instrumenting Java Bytecodes. In *The USENIX Symposium on Internet Technologies and Systems*, pages 73–82, 1997.
 - [12] International Telecommunication Union (ITU), X.509, The Directory - Authentication Framework, CCITT, 1989.
 - [13] A. Jøsang, Security protocol verification using SPIN, in J-Ch. Gregoire, editor, *Proceedings of the First SPN Workshop, INRS-Telecommunications*, Montreal, Canada, 1995
 - [14] Leveson, N. G., *Safeware: System Safety and Computers*, Addison-Wesley Pub. Co. Inc., 1995.
 - [15] Leffingwell, D., Widrig, D., *Managing Software Requirements: A Unified Process*, Addison Wesley Longman Inc., 2000.
 - [16] J. Lilius and I. Porres Paltor. vUML: a tool for verifying UML models. Technical Report 272, Turku Centre for Computer Science, 1999.
 - [17] Johan Lilius and Iván Porres Paltor, The Production Cell: An Exercise in the Formal Verification of a UML Model, TUCS Technical Report No. 288, May 1999
 - [18] D. Luckham and F. vonHenke. An Overview of Anna, a Specification language for Ada. *IEEE Software*, 2(2):9-22, March 1985.
 - [19] Manna, Z., Pruehli, A., *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag 1992.
 - [20] Mazza, C., Fairclough, J., Melton, B., De Pablo, D., Scheffer, A., Stevens, R., *Software Engineering Standards*, Prentice Hall, 1994.
 - [21] Robinson, W.N., Pawlowski, S., Managing Requirements Inconsistency with Development Goal Monitors, IEEE, *Transactions on Software Engineering*, November/December, 1999.
 - [22] Robinson, W.N., Volkov, S., Supporting the Negotiation Life-Cycle, ACM, *Communications of the ACM*, May 1998, pp. 95-102.
 - [23] Rosenschein, J., Zlotkin, G., *Rules of Encounter*, The MIT Press, 1994.
 - [24] Sergey Butkevich, Marco Renedo, Gerald Baumgartner, and Michal Young, "Compiler and Tool Support for Debugging Object Protocols, 14 pp. (OSU-CISRC-3/00-TR10), Ohio State University.
 - [25] Douglas H. Steves, Chris Edmondson-Yurkanan, Mohamed Gouda, A Protocol for Secure Transactions, *Proceedings of the Second USENIX Workshop on Electronic Commerce (EC96)* Oakland, California, November 18-21, 1996.