

Tracking Component-Based Software

Jerry Gao, Ph.D., Eugene Y. Zhu, Simon Shim, Ph.D.

San Jose State University
One Washington Square
San Jose, CA 95192-0180
Email:gaojerry@email.sjsu.edu

Abstract

Component engineering is gaining substantial interest in the software engineering community. A lot of research efforts have been devoted to analysis and design methods for component-based software. However, only very few papers address the testing and maintenance problems of component-based software. The paper focuses on the component traceability, related issues and solutions in testing and maintenance of component-based software. To increase component traceability in a distributed program, we propose a systematic approach to adding the component traceability in a distributed component-based program so that we can check, track, and monitor the diverse component behaviors and their performance in a distributed environment. Moreover, we introduce the concept of traceable components, and use examples to demonstrate how to construct them in a systematic way.

Keywords: Component engineering, software maintenance, software traceability, software engineering, component traceability.

1. Introduction

As the rapid increase of software programs in the size and complexity, it is very important to reduce high software cost and complexity while increasing reliability and modifiability [8]. With the advances of Internet technology, more distributed systems are built to meet diverse application needs. Currently component engineering is gaining substantial interest in the software-engineering community. As more third-party software components are available in the commercial market, more software workshops start to use the component engineering approach to developing component-based programs for distributed applications.

Although there are many published articles addressing the issues in building component-based programs, very few papers address the problems and challenges in testing and maintenance of software components and distributed component-based programs [11][9][2][10].

According to [2][9][10], engineers in the real world have encountered a number of issues in testing and maintenance of component-based software. Some of them related to component traceability and program tracking [2]. They are given as follows.

- *Hard to understand component behaviors in a system.* In system testing and maintenance, system testers usually have the difficulty in understanding and monitoring the component behaviors in a system due to the following reasons:
 - Engineers use ad hoc mechanisms to track the behaviors of in-house components. This has caused problem for system testers to understand the behavior of a system due to inconsistent trace messages, formats, and diverse tracking methods.
 - No built-in tracking mechanisms and functions in third-party components for monitoring their external behaviors.
 - No configuration functions for component clients to control and configure the built-in tracking mechanisms.
 - No systematic methods and technologies to control and monitor the external behaviors of the components.
- *Difficulty on component error isolation, tracking and debugging.* In a system,

components that are developed by different teams may use different tracking mechanisms and trace formats. Thus, inconsistent tracking mechanisms and trace messages cause the difficulty in error detection and isolation of components.

- *High costs on performance testing and tuning for components.* Performance testing for component-based programs is a major challenge in system testing due to the fact that current component vendors usually do not provide users with any performance information. Hence, system testers and integration engineers must spend a lot of efforts to identify the performance problems and the components that cause the problems.
- *Lack in system resource validation for components.* Since most components do not provide their system resource information, it is difficult for system testers to locate the system resource problems in system testing and maintenance.

Therefore, there are two major challenges in developing distributed component-based software. The first is to design software components with consistent mechanisms and interfaces to support the tracking of their behaviors, functions, performance, and resources. The other is to develop a systematic method and environment to monitor and analyze component behaviors and system performance in a distributed environment.

This paper addresses the component traceability issues in supporting of component-based programs. It discusses the component traceability concept, requirements, challenges, and evaluation criteria. Moreover, it examines different program tracking mechanisms. We introduce the concept of traceable components, and propose a new tracking mechanism called event-based tracking model. It provides a systematic mechanism for tracking various software components in a component-based program. According to the result of our prototyping implementation, this approach has shown its potential advantage on increasing component traceability in a distributed system with minimum programming efforts, a fixed system overhead, and performance impacts.

The structure of the paper is organized as follows. Section 2 discusses the perspectives of component traceability in terms of component behaviors, interfaces, performance, events, and status. In addition, it discusses the tracking requirements and evaluation criteria. Section 3 examines different mechanisms to increase component traceability. Moreover, it introduces the concept of traceable components and proposes an event-based tracking mechanism for software components in a program. Some application examples are given to demonstrate the basic idea. Section 4 describes a tracking environment to support components in a distributed program. Finally, the conclusion remarks are given in Section 5.

2. Understanding of Component Traceability

According to IEEE Standard Directory of Electrical & Electronics Terms, "tracking" refers to the process of following a moving object or a variable input quantity, using a servomechanism [3]. A trace routine refers to a program routine that provides a historical record of specified events in the execution of a program.

Software tracking includes a) *project tracking*, b) *product tracking*, and c) *program tracking*. *Project tracking* keeps tracking of important project schedules, activities and events during the software development process. It is a fundamental activity in software management [6][5]. *Product tracking* refers to control and monitor software product in a systematic and manageable way [7]. It is the essential task in configuration management. *Program tracking* refers to the activities and effort on tracking various factors of a program, including its input data, output results, and behaviors. It is an effective way to support engineers in program understanding, debugging, software testing and maintenance.

Component Traceability

According to Schmauch Chareles H., "traceability" refers to the ability to show, at any time, where an item is, its status, and where it has been [8]. "Traceability" of a software component refers to the extent of its build-in capability of tracking the status of component attributes and component behavior. Traceability of a component includes the following two aspects:

- *Behavior traceability* - It is the degree to which a component facilitates the tracking of its internal and external behaviors. There are two ways to track component behaviors. One is to track internal behaviors of components. This is useful for white-box testing and debugging. Its goal is to track the internal functions, internal object states, data conditions, events, and performance in components. The other one is to track external behaviors of components. It has been used for black box testing, integration, and system maintenance. The major purpose is to track component public visible data or object states, visible events, external accessible functions and the interactions with other components.
- *Trace controllability* - It refers to the extent of the control capability in a component to facilitate the customization of its tracking functions. With trace controllability, engineers can control and set up various tracking functions such as turn-on and turn-off of any tracking functions and selections of trace formats and trace repositories.

We can classify component traces into five types:

(1) *Operational trace* - It records the interactions of component operations, such as function invocations. It can be further classified into two groups: a) *internal operation trace* that tracks the internal function calls in a component, and b) *external operation trace* which records the interactions between components. *External operation trace* records the activities of a component on its interface, including incoming function calls, and outgoing function calls.

(2) *Performance trace* - It records the performance data and benchmarks for each function of a component in a given platform and environment. *Performance trace* is very useful for developers and testers to identify the performance bottlenecks and issues in performance tuning and testing. According to performance traces, engineers can generate a performance metric for each function in a component, including its average speed, maximum, and minimum speed.

(3) *State trace* - It tracks the object states or data states in a component. In component black box

testing, it is very useful for testers to track the public visible objects (or data) of components.

(4) *Event trace* - It records the events and sequences occurred in a component. The event trace provides a systematic way for GUI components to track GUI events and sequences. This is very useful for recording and replay of GUI operational scenarios.

(5) *Error trace* - It records the error messages generated by a component. The error trace supports all error messages, exceptions, and related processing information generated by a component.

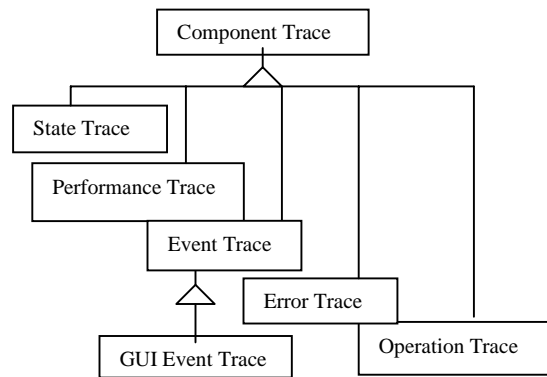


Figure 1. Different Types of Component Traces

Essential Requirements for Program Tracking

According to our observations, there are several basic requirements for component tracking and program tracking.

REQ1: Use standardized trace format and understandable trace messages. This is essential to generate trace messages that are easy to understand. The trace message format should be defined to include proper information to engineers in problem isolation, program understanding, and behavior checking. For example, each trace message must contain proper identification information, such as component identifier, thread identifier, object id, and class function name. All error messages and exception handling must be classified and defined with a fixed format.

REQ2: Minimize the engineers' efforts to add or remove tracking code in programs. According to our experience, this is a critical factor to a success of the usage of tracking mechanisms. Engineers and managers have realized that building the tracking

capability and functions in software components is necessary for testing and maintenance of component-based software. It is also an effective way to reduce the development cost. Since they are concerned about how much effort they must spend, some of them may hesitate to use a provided tracking mechanism unless it is easy to use and effective for debugging and performance testing.

REQ3: Provide flexibility on selecting different trace types. Since each trace type has its special usage on testing and maintenance of a component-based program, it is important to provide diversified trace types and facilities for engineers to use according to their needs.

REQ4: Control the system overhead involving in tracking. There are two types of system tracking overheads: a) system CPU time, and b) extra system resources, such as system space for tracking code, and storage space for trace messages. The primary issue is how to add the tracking capability in components with a minimum system overhead.

REQ5: Manage and store the selective trace messages. Engineers need this function in debugging, system testing and maintenance to help them manage and monitor the trace messages.

Evaluation of Component Tracking Capability

To evaluate the tracking capability of software components in a system, we define the following four levels:

Level 1: Initial – Software components are constructed with ad hoc tracking mechanisms, trace formats, and functions. For components at this level, engineers usually found that they need to spend more time in understanding, debugging, and testing during system testing and software maintenance.

Level 2: Standardized – Software components are built according to a pre-defined standard tracking mechanism, trace format, and a set of pre-defined functions. Using standardized tracking in component construction enhances the understanding of component behaviors in distributed software, and reduces a lot of costs on debugging and system testing. However, there is a programming overhead cost involved in adding built-in tracking code into components.

Level 3: Systematic – At this level, software components are designed with a systematic tracking mechanism and capability, therefore, engineers can easily to monitor and check their behaviors, and manage and store their traces in an automatic manner. Since a consistent tracking mechanism is used for building traceable components, engineers' programming efforts involved in tracking is reduced.

Level 4: Customizable – Software components are designed to facilitate the support of component tracking functions and their customization. Achieving customizable tracking in component construction provides the flexibility to define, select, and configure the tracking functions and trace formats. Moreover, it helps engineers to set up a systematic tracking and monitoring method and environment for supporting component-based software.

3. Mechanisms to Increase Component Traceability

Since a component-based program is an integration of software components, its program traceability depends on the traceability of its components. Component observability depends on component traceability [2]. In the real word practice, we have used ad hoc mechanisms or pre-defined facilities to add tracking code into programs to increase program understandability. However, we have encountered the difficulty on supporting diverse in-house components and commercial components (COTS). This section first discusses three different systematic tracking mechanisms and examines their pros and cons. Then, we define the concept of a *traceable* component, and describe its essential functions and architecture. Moreover, we propose a new event-based tracking mechanism based on traceable components.

Table 1 lists three basic approaches to add a consistent tracking capability into software components to increase component traceability, and to enhance program.

Method 1: Framework-based tracking – In this approach, a well-defined tracking framework (such as a class library) is provided for component engineers to add program tracking code according to the provided programming reference manual. It usually is implemented based on a trace program

library. Component engineers can use this library to add tracking code into components. This approach is simple and flexible to use. It can be used to support all trace types, especially for error trace and GUI trace. However, there are several drawbacks. Firstly, it requires a high programming overhead. Secondly, it relies on engineers' willingness to add tracking code. Moreover, this approach assumes that component source code is available. Therefore, it is difficult to deal with commercial components (COTS) because usually they do not provide any source code to clients.

Table 1. Comparisons of Different Tracking Mechanisms

Tracking Perspectives	Framework-Based Code Insertion	Automatic Code Insertion	Automatic Component Wrapping
Source Code	Needed	Needed	Not needed
Code Separation	No	No	Yes
Overhead	High	Low	Low
Complexity	Low	Very High	High
Flexibility	High	Low	Low
Applicability	All types	OP trace, performance trace	OP trace, performance trace
Applicable Components	In-house components	In-house components	In-house components and COTS

Method 2: Automatic code insertion – This approach is an extension of the previous one. Besides a tracking framework, it has an automatic tool, which adds the tracking code into component source code. A parser-based tracking insertion tool is a typical example. To add operational tracking code, it inserts the operation tracking code into each class function at the beginning and the end of its functional body to track the values of its input data and output parameters. Similarly, it can insert the operation tracking code before and/or after each function call. Performance tracking code can be added into a component in the same way to track the performance of each function. Although this approach reduces a lot of programming overhead, it has its own limitations. First, this approach assumes that component source code is available. This causes the limitation of using it on commercial components (COTS) due to the lack of source code. Next, it is not flexible to insert diverse tracking code at any place in components due to its automatic nature. The other problem is its complexity of the

parser tool. Since a component-based program may consist of components written in different languages, this causes high complexity and costs on building parser tools.

Method 3: Automatic component wrapping - This approach is another extension of the first one. Unlike the second method where tracking code is inserted by parsing component source code, this approach adds tracking code to monitor the external interface and behaviors of components by wrapping them as black boxes. The basic idea is to wrap every reusable component (or third-party component) with tracking code to form an observable component in the black box view. With the tracking code, engineers can monitor the interactions between a third-party component and its application components. This approach is very useful to construct component-based software based on third-party software components, for example, EJB. Compared with the other two methods, it has several advantages. One of the advantages is its low programming overhead. In addition, it separates the added tracking code from component source code. Since no source code is required here, this method can be used for both in-house reusable components and commercial components (COTS). However, it is not suitable to support error tracking and state tracking because they are highly dependent on the detailed semantic logic and application domain business rules.

It is clear that each approach has its own pros and cons. In real practice, we need to use them together to support different types of tracking for a program and its components. To design and construct traceable components, engineers need more guidelines on component architecture, tracking interface, and supporting facilities. They encounter the following challenges:

- (1) How to design and implement traceable and observable components in a distributed system?
- (2) How to provide efficient and effective tracking mechanisms with minimum programming effort and system overhead?

In the rest of this section, we have made our attempt to answer these questions by providing a new tracking mechanism called event-based tracking model. The basic idea is similar to Java event model. We consider all types of tracking requests in software components as tracking events. The

software components and their elements, which issue tracking requests, are tracking event sources.

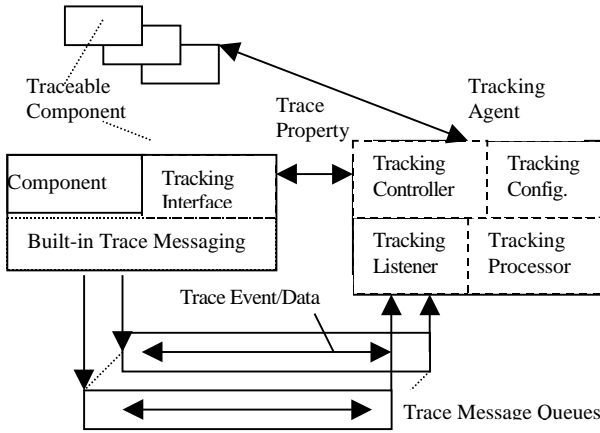


Figure 2. Structure of Event-Based Tracking Model

Event-Based Tracking Model

The event-based tracking model is a systematic mechanism that supports engineers to monitor and check the behaviors of components and their interactions in a component-based program. Its basic idea is influenced by Java event model for GUI components. We consider all software components and their elements as *tracking event sources*. In a software component, component engineers or an automatic tracking tool can add built-in tracking code to trigger five types of tracking events. They are performance tracking, operational tracking, error tracking, state tracking, and GUI tracking events. These events are packed as tracking event messages, and added into five trace message queues according to their types. To catch different tracking events, we use a *tracking listener* to receive these events, dispatch them to a *tracking processor* to generate the proper trace, and store them into a specified trace repository. As shown in Figure 2, the event-model tracking mechanism relies on a *tracking agent* to support a collection of *traceable components* in a computer. Intuitively, a *traceable component* is a software component, which is designed to facilitate the observation and monitor its behaviors, data and object states, function performance, and interactions to others. A *tracking agent* consists of the following four functional parts.

- *Tracking Controller* serves as a controller of the agent to interact with traceable components. It performs component tracking registration, discovers component trace property, and sets up a *tracking*

listener for each traceable component. Figure 3(a) shows the set-up interaction sequences between a traceable component and a tracking controller.

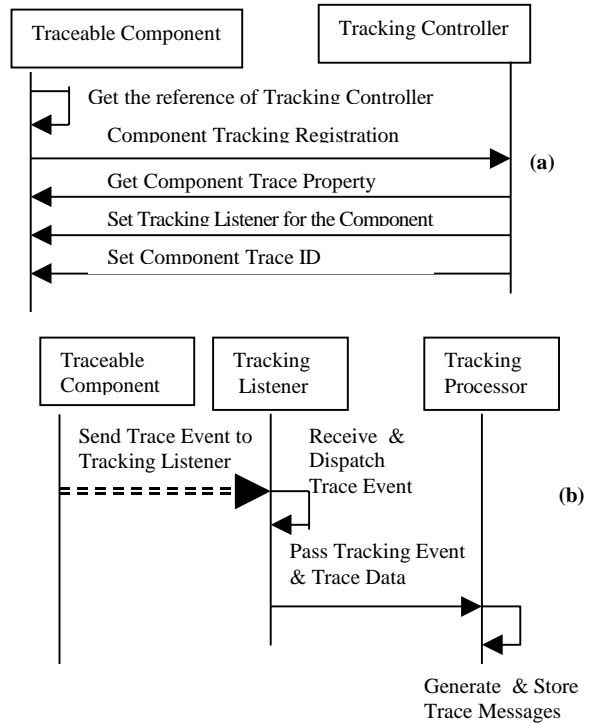


Figure 3 Interaction Sequences

- *Tracking Listener* is a multi-thread program that listens and receives all types of tracking events through five trace message queues, and dispatches them to *tracking processor*. Figure 3(b) shows the interaction sequences among a traceable component, tracking listener, and tracking processor

- *Tracking Processor* generates program traces according to a given trace event based on its trace type, trace message and data, and stores them in the proper trace repository.

- *Tracking Configuration* allows a user to discover and configure various tracking features for each traceable component.

In the event-based tracking model, a tracking agent communicates with all traceable components through trace message queues. To explain the mechanism, we use Java Message Queue and Java classes to demonstrate the details of the mechanism. Figure 4 shows two Java classes, *TraceMessage* and *TraceMessageQueue*. Figure 5 provides the logic of a *tracking listener*. A tracking controller can use

several *tracking listeners*. Each of them is waiting and processing one trace event message queue.

```

public class TraceMessage {
    public int traceID;
    public long timeStamp;
    public int traceType;
    public Byte[] traceBuffer;
    public TraceMessage(int sizeofBuffer){
        traceBuffer = new Byte[sizeofBuffer];
    }
}
import TraceMessage;
public class TraceMessageQueue {
    public void send(TraceMessage msg){
        // send the message to the message queue
        .....
    }
    public void receive(TraceMessage msg){
        ..... // receive the message from the queue
    }
}

```

Figure 4 Trace Message and Queues

```

import TraceMessageQueue;
import TraceMessage;

public class TraceListener extends thread {
    private TraceMessageQueue msgQueue;
    public TraceListener(TraceMessageQueue q) {
        msgQueue = q;
    }
    public void run() {
        for(;;) {
            // receive message from the message queue
            TraceMessage msg = msgQueue.receive()
            ..... // process the trace event message and
            ..... // pass trace events & data to tracking processor
        }
    }
}

```

Figure 5. A Trace Listener Example

A software component is *traceable* if it is constructed in a way to add component tracking capability to support the monitoring of its behaviors in a systematic manner. In our view, a traceable component contains three parts: a) essential functional parts, b) *standard component tracking interface*, and c) *standard built-in trace messaging*.

```

import java.util.*;

public interface TraceableComponentInterface {
    public String getComponentName(); // return component name
    public Vector getTraceProperties(); // expose the trace source
    public void setTraceQueues(Vector queueList); //set queues
    public void setTraceID(int TraceID); //set the unique trace id
    public void bindTraceController(); // establish the connection
        // between the component and the tracking controller
}

```

Figure 6 Traceable Component Interface

A *component tracking interface* in a traceable component provides a standard built-in interface for tracking. With this interface, traceable components interact with a *tracking controller*. As shown in Figure 6, a component tracking interface must include the following functions:

(1) *Set-up trace* - These interface functions set up the trace message queues and trace ID. *SetTraceID(int TraceID)* and *setTraceQueues(Vector queueList)* in Figure 6 are typical examples.

(2) *Discover trace properties* - These interface functions allow a component tracking controller or other clients to discover the trace sources and trace properties in a component. The typical examples, of trace properties are trace types, trace names, and set-up parameters. *GetComponentName()* and *getTraceProperties()* listed in Figure 6 are two examples. Figure 7 shows a trace property class that defines the basic attributes of a trace property, including trace type, name, parameter. The component clients can define them their values when they implement a traceable component interface. Figure 9 shows an example how to set up the properties for an operation trace type by defining *getTraceProperties()*.

(3) *Connect to tracking controller* - This interface function can be used to connect the traceable component to a *tracking controller* for its registration. *bindTraceController()* in Figure 6 is an example. A detailed implementation of this function is given in Figure 9.

After connecting to a *tracking listener* through the *tracking controller*, all trace events occur in the component will be reported to the trace listener in asynchronous mode through trace message queues.

```

public class TraceProperty {
    public int traceType; // trace type:
    // 1 = T_OPERATION_TRACE, 2 = T_INTERFACE_TRACE
    // 3 = T_PERFORMANCE_TRACE, 4 = T_STATE_TRACE
    // 5 = T_ERROR_TRACE, 6 = T_GUI_CONTROL_TRACE
    public String traceName; // trace name
    public String traceParameter; // Additional parameters for the trace
}

```

Figure 7 Trace Property Class

Standard built-in trace messaging refers to the program codes inserted in a component to send various tracking events and messages to a bound

tracking listener. After a *tracking listener* receives and processes them, it passes them to the *tracking processor* to generate trace messages according to a pre-defined format in a trace repository.

Application Examples

There are three ways to write traceable components based on this idea. To demonstrate our idea, we use a Java class in Figure 8 as a component example. The given class has no built-in tracking capability. To make it as a traceable component, we need to do two things. The first is to implement the traceable component interface in Figure 6. Figure 9 shows the *TraceEngine* class, an example of the implementation, where *setTraceMessageQueues(..)*, *setTrace(..)*, *getTraceProperties()*, and *bindTraceController()* is defined in details. Next, we can build a traceable component based on the *TraceEngine*. Figure 10 shows a wrapped traceable component using a wrapping approach. This can be done in a systematic way using EJB container idea. Figure 11 shows a traceable component that is constructed by adding tracking code inside the given component. This can be done manually or automatically.

```
public class SimpleComponent
{ // constructor
  public SimpleComponent() {
    // constructor body
  }
  // private data members
  private String Member1;
  // the property functions
  public void setMember1(String member) {
    Member1 = member;
  }
  public String getMember1() {
    return(Member1);
  }
  public void function1() {
    ..... // the function body
  }
}
```

Figure 8 A Simple Component Example

Summary of Event-Based Tracking Model

In short, this approach can be used to support all trace types. Since engineers can use different ways to add tracking code into software components, it is very useful and flexible. Because the event-based tracking model is developed based on well-defined traceable components. It can be easily implemented to support component-based software. It has several distinct features. Firstly, it defines a standard tracking interface to support various components.

Secondly, the programming overhead for built-in tracking code is reduced because of its systematic nature. Finally, it reduces the system overhead involved in tracking because a *tracking agent* takes care of tracking configuration, track event listening, and processing. Since the communications between a *tracking agent* and traceable components are performed in an asynchronous mode, the system performance and resources are

```
import TraceControllerInterface;
import TraceProperty;
import TraceMessage;

public class TraceEngine
  implements TraceableComponentInterface {

  // implement the traceable component interface
  private Vector tMQueues;
  private int traceID;
  public String getComponentName() {
    return ("TraceableComponent"); }

  public void setTraceMessageQueues(Vector queueList) {
    tMQueues = queueList; }
  public void setTraceID(int TraceID) {
    traceID = TraceID; }
  public Vector getTraceProperties() {
    Vector traceProperties = new Vector();
    // create and add the trace elements to the vector
    TraceProperty tp = new TraceProperty();
    tp.traceType = 1;
    tp.traceName = "functionName";
    tp.traceParameters = "";
    // the function functionName contains 0 arguments.
    traceProperties.addElement(tp);
    .....
    return(traceProperties);
  }
  public void bindTraceController() {
    TraceControllerInterface tController;
    // obtain the reference of the trace controller.
    .....
    // register the component to the trace controller
    tController.registerComponent(this);
  }
  public void send(TraceMessage msg)
  {
    (TraceMessageQueue)
      tMQueues.elementAt(msg.traceType).send(msg);
  }
}
```

Figure 9 An Implemented Trace Interface

4. Tracking Environment for Distributed Component-Based Systems

In this section, we describe a tracking environment for distributed component-based systems. It uses our event tracking model to implement a systematic program tracking

mechanism in software components to achieve program tracking in a distributed environment.

```

import TraceEngine;
import TraceMessage;
public class TraceableComponent extends SimpleComponent
{ TraceEngine tracker; // the reference to the trace engine
  // wrapped constructor
  public TraceableComponent() {
    super();
    tracker = new TraceEngine();
    tracker.bindTraceController();
  }
  public void function1() { // wrapped function member
    // create the trace message
    TraceMessage msg = new
    TraceMessage(OPERATION_TRACE_SIZE);
    ..... // pack the trace message
    // send the message
    tracker.send(msg);
    super.function1(); // call original function
  }
  ..... // other wrapped function members
}

```

Figure 10 A Traceable Component Example

```

import TraceEngine;
import TraceMessage;

public class TraceableComponent
{
  TraceEngine tracker;
  // the reference to the trace engine
  // constructor
  public TraceableComponent() {
    // original constructor body
    .....
    tracker = new TraceEngine();
    tracker.bindTraceController();
  }
  public void function1(){
    // create the trace message
    TraceMessage msg = new
    TraceMessage(OPERATION_TRACE_SIZE);
    ..... // pack the trace message
    ....// send the message
    tracker.send(msg);
    ....// original function body
  }
  ..... // the rest of original function members
  ..... // original data members
}

```

Figure 11 A Traceable Component Example

As shown in Figure 3, the environment consists of a number of *tracking agents* and a *tracking server*. For a distributed component-based system, each machine on the network has a *tracking agent*. The agent interacts with traceable components deployed on the same machine to control, record, and monitor

diverse component behaviors, and generate the trace data based on a given message format. The tracking server plays a role of a central server to allow engineers to control tracking agents to collect diverse trace data and analyze them.

Tracking Agent

As seen in Figure 12, a tracking agent has the four parts: a) a *trace listener*, b) a *tracking configuration module*, c) a *tracking processor*, and d) a *communication interface*. The *trace listener* interacts with traceable components to listen to different tracking events happened on the trace sources defined in the components, and passes them to the *tracking event engine*. The *tracking configuration module* is a GUI interface that supports engineers to select and configure trace types and turn on and turn off related tracking functions. The *tracking processor* handles different tracking events (or requests) to record corresponding trace data and/or messages in a local *trace repository*. The *communication interface* interacts with the *tracking server* to perform three functions: a) agent registration, b) execution control, monitor, and c) trace data (or message) transfer.

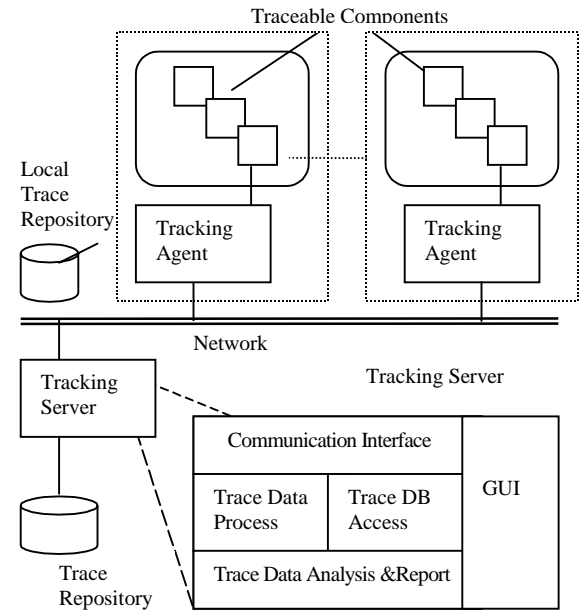


Figure 12 Distributed Tracking Environment

Trace Format

Using a standardized trace format is essential to generate consistent trace messages for each component in a distributed program. A well-defined

trace format (see Figure 13) facilitates the generation of understandable trace messages. This increases component understanding, and helps error isolation and bug fixing. Our distributed tracking environment supports two types of information:

- *Trace commands* - Trace commands support the tracking server to control and communicate with tracking agents in the distributed environment. Each command includes command message ID (00), command code, time stamp, and command parameters.
- *Trace data* - Trace data include message ID (01), trace type, time stamp, trace data, and trace identifier. Trace data indicates where and when the trace message is generated. A trace identifier includes component identifier, thread identifier, and object identifier. Each trace type has its specific trace data format. Tracking agents generate diverse trace data for each traceable component in a local trace repository.



Figure 13 Trace Message Format

5. Summary and Conclusions

In this paper, we introduce the concept of *traceable components*, demonstrate how to make components traceable so that engineers can easily monitor and check various behaviors, status, performance, and interactions of components in system testing and maintenance supported. Moreover, we propose a new systematic method (*event-based tracking*) to help engineers set up a systematic environment to support and monitor component behaviors of a component-based program in a distributed environment.

We have developed and implemented the essential parts of the mechanism in our Distributed Supporting System for Components (DSSC) using Java, Java Bean, Java Message Queues, and EJB. Our application results show that the mechanism is not only feasible, but also effective and useful to track various types of component behaviors.

The major advantages of this mechanism include: a) low costs on programming effort, b) flexible to use, configure, and change, c) consistent trace format and tracking code, d) applicable to both in-house components and COTs, and e) very little impact on component performance and functions. We are currently investigating new trace analysis models, methods and tools to support fault detection, fault isolation, and performance analysis.

6. References

1. Bernard, Londeix, Cost Estimation for Software Development, New York: Addison-Wesley, 1987.
2. Gao, Jerry, "Testing Component-Based Software", STARWEST'99.
3. IEEE Standard for Software Verification and Validation, New York, 1998.
4. Jirotko M, Goguen J, "Requirements Engineering", Academic Press, 1993.
5. Kit Edward, *Software Testing in the Real World*, New York: Addison-Wesley, 1995.
6. McConnell, Steve. "Rapid Development: Taming Wide Software Schedules", Microsoft Press, 1996.
7. Rada Roy, "Reengineering Software - How To Reuse Programming To Build New State-of-art Software", 2nd edition, New York: Intellect Ltd., 1999.
8. Schmauch Charles H., ISO 9000 For Software Development: Revised Edition, Wisconsin: ASWC Inc., 1995.
9. Voas, J., "Maintaining Component-Based Systems", IEEE Software, Vol. No. 1999.
10. Voas, J. M. and Miller, Keith W., "Software Testability: The New Verification", IEEE Software Vol. 12, No. 3: May 1995, pp. 17-28.
11. Weyuker, Elaine J, "Testing Component-Based Software: A Cautionary Tale", *IEEE Software*, September/October 1998.