

# An Instrumentation Engine for Dynamic Program Analysis

Atul U. Nulkar

Computer Science Department  
Colorado State University  
Fort Collins, CO 80523

(970) 491-6228  
nulkar@cs.colostate.edu

Roger T. Alexander

Computer Science Department  
Colorado State University  
Fort Collins, CO 80523

(970) 491-7026  
rta@cs.colostate.edu

## ABSTRACT

We describe our present on-going research work on a software tool called the *Dynamic Instrumentation Engine (DIE)*. The main goal for designing and developing the DIE is to aid in dynamic program analysis and instrumentation without modifying the source code and thus ease the process of information gathering activities associated with instrumentation and testing of software applications. This form of instrumentation is on-the-fly and non-invasive since it has no computational side-effects. The DIE should prove to be an excellent software tool to gather valuable information about a program under analysis, for areas such as dynamic program analysis and data collection, temporal assertion monitoring, test coverage measurement, etc.

## Keywords

Program Instrumentation, Dynamic Analysis, Temporal Assertion Monitoring, Test Coverage Measurement, Program Analysis

## 1. INTRODUCTION

Instrumentation is the process of inserting additional statements into a program for the purpose of gathering information about its dynamic behavior [1]. Applications of program instrumentation include test-coverage measurement, test-case effectiveness assessment, assertion checking, dataflow-anomaly detection and pathwise decomposition [1].

Traditional program instrumentation is carried out by manually inserting instruments in the source code. This source level instrumentation can result in undesirable side effects on the original program  $P$ , thus altering  $P$ 's intended behavior at runtime.

Ideally, we want the output of  $P$ 's instrumented version,  $P'$ , to be a superset of  $P$ 's output. That is, the instrumentation added to  $P$  should not have an effect on the computation of  $P$ . Unfortunately, it is often very difficult to have insight into the effects that a particular instrumentation will have on the execution state of  $P$ . Often the effects of executing a particular piece of instrumentation  $I$  are not obvious by merely examining the presence of  $I$  in the source code of  $P$ . For example, if  $I$  calls a standard library function  $f$ , that call may affect the instructions of  $P$  that follow the execution of  $I$ . What is needed is a way to instrument  $P$  without introducing side effects into  $P$ 's execution.

One solution is to have some form of non-intrusive instrumentation that executes under the control of a separate tool while  $P$  is executing, such that the instrumentation does not have

side effects on the state of  $P$ . This is the main goal and idea behind our Dynamic Instrumentation Engine (DIE). Another objective for the DIE is to tackle the problems of scalability and extensibility when instrumenting and testing large scale software applications. The DIE will not be tailored to any particular software technology, or a specific type of software application, but will support a wide variety of commonly used languages. The general requirement guiding our design and implementation of the DIE are that it must work for any type of instrumentation that can be done without modifying the program under test.

Our current research focuses on key aspects and implementation of Dynamic Program Instrumentation in the form of the DIE. It is how this would lead towards improved information gathering associated with a variety of software testing related activities, as mentioned in the abstract.

## 2. ARCHITECTURAL OVERVIEW

Figure 1 depicts the high level architecture of the DIE, including relevant interfaces, layers and modules. A guiding principle behind the architecture of the DIE is that it be language and platform independent.

The key insight behind the DIE architecture is that a program to be instrumented,  $P$ , can be executed under the control of a debugger. Given this insight, program instrumentation can be treated as an abstract entity that is separate from the physical source code characteristics of  $P$ , and instead be represented as a set of *instrumentation instructions*. Each instrumentation instruction  $I$  is described in a language that is independent of the language used to express the source code of  $P$ .  $I$  encodes a location  $l$  (i.e. statement in  $P$ ) that is to be instrumented, the instrumentation action to be carried out (e.g. tracing or intermediate state capture), and metadata that describes salient characteristics of  $l$  relevant to the objective of the instrumentation (e.g. variables to be monitored for state capture, or an identifier that uniquely identifies  $l$  in  $P$ ).

The initial implementation of the DIE is based on the Sun Java (J2SE) platform [2] and the Java Platform Debugging Architecture (JPDA) [3]. While this particular implementation is specific to Java programs, the DIE architecture can support other languages. This is achieved through separation of concerns by having a specific language platform represented as an abstract entity from DIE's perspective.

The DIE incorporates a layered architecture, as shown in Figure 1. The topmost layer of the DIE depicts the *Platform Debugging Architecture (PDA) Adapter Layer*. This layer presents an abstract interface that the DIE uses to communicate with and control a particular debugging platform (DP). Each PDA Adapter interacts directly with a particular kind of DP (the JPDA in our initial implementation) and with the DIE. A PDA Adapter's key responsibilities include receiving *logical control instructions* from the DIE and translating these into *physical control messages* and then forwarding them to the actual DP. The adapter is also responsible for receiving *physical execution events* and data from the DP, translating these to *logical execution events* and forwarding these to the DIE.

The DIE can handle varying kinds of instrumentation because it includes an interface for pluggable Program Instrumentation Modules (PIMs) (depicted as the lowest architectural layer in Figure 1). A custom PIM is built for each kind of instrumentation that will be deployed against a subject program *P*. Each PIM is implemented according to a well defined interface specified as part of the DIE architecture. The DIE uses this Interface to deliver logical execution events to each PIM as the execution of *P* progresses. In response, a PIM consumes each event and determines what *execution actions* the DIE should direct the PDA Adapter to take.

### 3. EXECUTION MODEL

The DIE accepts as input, a *DIE Configuration* description that is expressed in an XML schema, and contains instructions that specify how *P* is to be monitored by the DIE. *Job Configurations* contain instructions specific to the program, such as environmental conditions, runtime parameters, command line information, etc. Upon loading and interpretation for a particular configuration description, the DIE invokes a separate Java Virtual Machine (JVM) that executes a subject program under whatever environmental conditions specified by the *Job Configuration*. The *Job Configuration* identifies the various *Program Instrumentation Modules (PIMs)* using *PIM* specific job data. A *PIM* carries out a particular instrumentation action (e.g. monitoring *P* for branch coverage).

Each PIM is set up using its own *Module Configuration (Module Config)*. Modules can communicate with each other through the DIE. A Module registers itself with the DIE and includes the events of interest. After starting the separate JVM and before execution of *P* begins, the DIE configures the remote JVM to report debugging events of interest based on the *DIE Configuration*. *P* is then executed in the separate JVM.

As the events registered for *P* are received by the PDA Adapter, *P* is suspended and the occurrence of an event is reported to the DIE. Upon receipt of the event notification, the DIE delivers the event to each registered PIM, based on the event type, using a *message bus*. Each module interprets and consumes the event notification, and then returns control to the DIE, which the DIE passes the event to the next PIM in the chain. Optionally, a PIM can signal that no the event should receive no further processing.

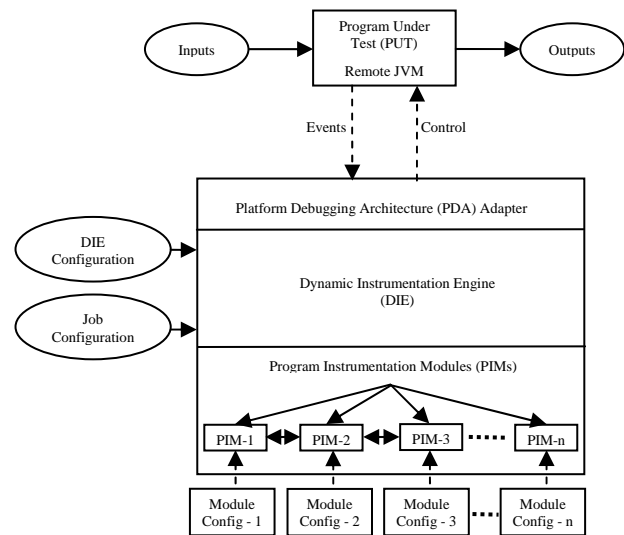


Figure 1

After processing of the event is complete, the DIE then performs the instrumentation action that is called for by the configuration instructions. The DIE then instructs the PDA Adapter to have *P*'s execution resumed, unless an instrumentation instruction or a PIM indicates that the job should terminate.

### 4. CURRENT AND FUTURE WORK

Our current work is focused on completing the implementation of the DIE along with several instrumentation modules. Once complete, our focus will turn to using the DIE for investigations in temporal assertion monitoring, dynamic analysis of design patterns, and fault injection.

### 5. CONCLUSIONS

We foresee many benefits from using the DIE to efficiently perform dynamic program analysis. We plan to design and develop the DIE for eventual release into the open source community to foster ideas, growth and development, and to achieve more scalability and extensibility. The DIE can potentially eliminate the need for static instrumentation based on troublesome and error prone source code modification, as well as extend support for virtually any type of instrumentation.

### 6. REFERENCES

[1] J.-c. Huang, Program Instrumentation and Software Testing. COMPUTER, 1978. 11(4).  
 [2] Java™ 2 Platform, Standard Edition (J2SE), <http://java.sun.com/j2se/>  
 [3] Java™ Platform Debugger Architecture, <http://java.sun.com/j2se/1.3/docs/guide/jpda/index.html>