Collecting the Inter Component Interactions in Multithreaded Environment

Arun Mishra, Alok Chaurasia, Pratik Bhadkoliya, and Arun Misra

Computer Science & Engineering, MNNIT,
Allahabad (U.P.), India
{rcs0802,cs074037,cs074003,akm}@mnnit.ac.in

Abstract. Whenever a new upgrade in software is made or a new component is loaded, that changes may impact the existing system's execution .Our objective is to define an approach that can be applied to on-line validation of component integration in autonomous system (AS). One important means of assuring the validation of component interactions is through analyze interactions among different components. Several tools have been developed to capture interactions among components. These tools do not support all the requirements for building interaction diagram in multithreaded environment. We have developed a technique to capture the run-time components interactions using .NET CLR mechanism. By this technique, we have been able to successfully capture the interactions among components across all application threads. A case study has been carried out on multithreaded self-adaptive system.

Keywords: Component Interactions, Runtime, Self-adaptive System, Multithreaded System, Trace.

1 Introduction

Component based system has been widely used in various application domains. However, lack of information about components developed by other developers and the complex interactions among components lead to validation challenges. As a matter of fact, consider the situation that 'CompB' calls 'Func2' of the 'CompC', while 'CompA' has already invoked 'Func1' of the 'CompC'. Similarly, 'Func3' of the 'CompD' gets a call from 'CompC'. In this situation it is difficult to recognize the thread that gave rise to the invocation of 'Func3' towards 'CompD'.

Multithreaded system cannot be depicted using standard UML representation as shown in Fig 1. This introduced the need for identifying the thread originator of the each event; our objective is to define an approach to visualize the inter component interactions in such component based multithreaded systems.

A general profiler works with CLR (middleware of .NET Framework) and collects all the traces of execution at the runtime. This type of profiling reduces the performance of the system by including unnecessary information (such as predefined .NET classes' interactions, intra-component interactions etc.).We have developed a technique to trace the component interactions at runtime, across all threads of the system with minimal overhead. The interactions thus traced are used to build 'Trace Diagram' which inherits UML sequence diagram standards and extends some rules.

N. Meghanathan et al. (Eds.): CCSIT 2011, Part I, CCIS 131, pp. 452–461, 2011.

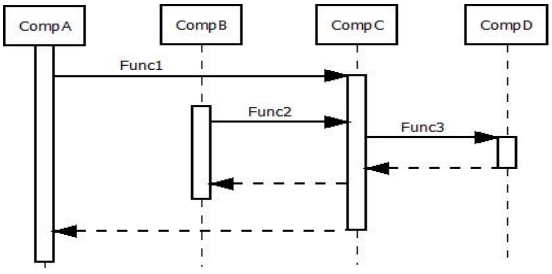

Fig. 1. Concurrent Method Invocation Problem

Our approach can be applied to runtime analysis of component integration in self-adaptive system. In the self-adaptive system, components are dynamically replaced by a similar component in terms of functionality, whenever the current working component leads to reduction of system's performance. A case study has been carried out on a component based multithreaded self-adaptive system in which the application uploads file to a server over Internet under its normal behavior. When bandwidth is found below a certain threshold value, the application adapts a new component called ZIPP that compresses the file and uploads compressed version of the file to the server. Multiple threads are running concurrently in the system for different tasks that includes observing and analyzing the bandwidth available to the application, uploading the file to server and for interaction with the user. The custom profiler and distiller are implemented in Visual C++ using .NET Profiling API. Traces are collected at runtime and stored into a trace file at the end of the execution and can be inspected. A trace diagram has been given in the end, highlighting the inter component interactions in multithreaded environment of the case study file transfer application.

The paper is organized in the following manner: section 2 presents the literature review in field of self-adaptive system and validation; section 3 presents our approach towards the objective; section 4 discusses the runtime profiling in terms of customized profiler and distillation; section 5 presents case study, in which we show the trace diagram; section 6 finds place for conclusion and future work.

2 Literature Review

Automated software (a kind of complex software) has been successfully applied to a diversity of tasks, ranging from strong image interpretation to automated controller synthesis [1] [2]. Due to complexity of analyzing the complex system (because of multithreaded or concurrent nature and the black box nature of components) several different approaches to monitor their behavior have been proposed. An approach used in [3], authors instrument the architectural description, and not the middleware and they require the developers to define a set of rules used to analyze the traces. Atanas R. et al. in their work introduced the UML based technique to generate the different aspect of component interactions after the modification [4] and used those results to define testing goals. Felipe C. et al. worked on reverse engineering environment to support extraction and detection of implied scenarios from dynamic system information captured during the execution of existing concurrent applications [5]. The main contribution of their work is a practical demonstration of applying filters to the set of

captured events. Atanas R. et al. generalize the technique for dynamic analysis of multi-threaded programs by keeping multiple traversal stacks [6]**.** Giovanni M. et al. gave the idea to register the invocation and return timestamps of methods calls between the components [7]. Their tool relies on dynamically instrument methods during class loading.

3 The Approach

Multithreaded system includes several threads which are executed concurrently. The problem comes in visualizing the exact behavior of the system under various circumstances. Our approach lies in tracing the runtime components interactions on the basis of runtime customized profiling, distillation. Custom profiler works along with distiller and is responsible for the collection of traces at runtime for the component based multithreaded system. These traces are interpreted using trace diagram. Multithreading increases the degree of complexity in the system.

To demonstrate the concurrent method invocations in case of multithreading system we extend the UML standard in order to capture an execution trace that can be represented using a trace diagram as shown in Fig 2. This solution successfully overcomes the problem present in Fig 1.

Following Rules represents the extended rules to explain the process in multithreaded environment.

- Different colors are used to represent each thread, Object activation lanes and method calls and returns.
- The period of time in which we do not know what the thread is doing is represented with a vertical waved line from the time of return arrow starting with temporal gap to the time of the first known method call.
- Thread start and end are marked with a filled circle.

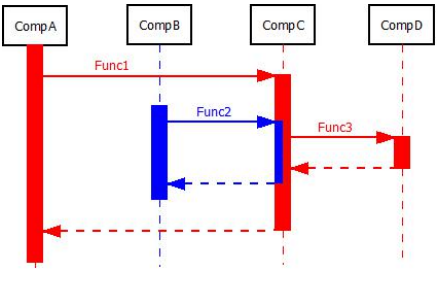

Fig. 2. Concurrent Method Invocation Solution

4 Runtime Profiling

Runtime profiling involves the technique to construct the exact execution flow of each thread. For the profiling purpose we plan to trace information about the behavior of the system during its execution. To profile the system at the runtime we need profiling activity at the middleware level [8].

In proposed work, profiler directly works with CLR (middleware of .NET). CLR provides logical place to insert hooks to see exactly what the system up to [9]. Depth of the result obtained after profiling is too high; so, we designed a customized profiler that includes distillation process.

4.1 Customized Profiler

We designed a customized profiler to profile the system, when its components are loaded at run time. By exploiting a hook provided by the CLR, it is possible to register our customized profiler that impacts each interaction and reports only the events of our interest in multithreaded environment. This approach provides the user with high level of transparency. This approach has very low overhead between the events and its tracing. In order to reduce the impact of information overload, customized profiler has the distillation on the events.

4.2 Distillation

The reason for us to use distillation process for the profiling is that our interests are confined only to inter-component interactions and not in intra-component interactions. Boundary calls allows us to distill lot of intra-component interactions. A boundary method call is a call whose receiver if lies in component A, then its caller cannot be component A. Boundary calls facilitates us to reduce the state space of method call combinations. We apply three types of distillers.

- Methods call belong to utility classes
- Calls to class constructor
- Calls to inner methods.

These distiller types, when applied to the execution traces gives the different levels of abstraction. At the first level we intend to better confine the system by preserving only those method calls that are directly involved in its implementation. At this level profiler distills out all method calls belonging to utility classes i.e. classes that offer common services to several other classes of the target system. The second and third levels distilled out further method calls considered irrelevant from the perspective of

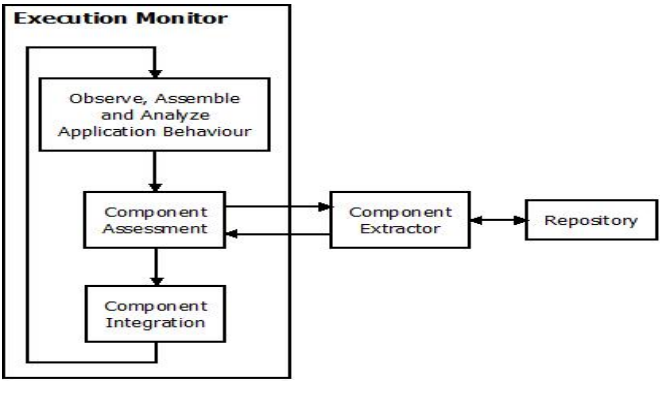

Fig. 3. Framework for Component Based Self Adaptive System

components interactions, namely, invocation of class constructors and invocations of inner methods, respectively. Distillation is intended to better confine the target system under analysis. These distiller types can vary as per user's requirements and need.

5 Case Study: Applying the Approach to the Component Based Multithreaded Self-Adaptive System

In self-adaptive systems where condition for the execution is changing continuously, component performance may degrade dramatically and may cause reduction of system performance. In our previous work [10], we have introduced a framework that automates the component extraction from repository and integration process. We have presented a basic tree equivalency algorithm to compare components.

Fig 3 represents a framework for component based self-adaptation at run-time [10]. Framework is represented as a collection of integrated components, allowing run-time adaptability.

The 'Execution Monitor' continuously monitors the behavior of the application while it is running. The major functions of the system are:

- Observe, assemble and analyze application behavior concurrently with application execution in a different thread.
- If diversion is found with respect to expected behavior of application, the adaptation strategies must reason about the impact of diversion and extract an appropriate component from the 'Repository', with the help of 'Component Extractor' that suits the current context.
- Integration of the new component that includes loading the new one, unloading the old (faulty) one, and restarting services to assure the functionality of the application.

5.1 Test Application

File transfer over internet is a day to day job for most computer users. The network bandwidth available to a computer node in a private network connected to internet, changes significantly due to request overhead by other computer nodes in the private network. When bandwidth available is low, file transfer takes longer time than usual. One approach to overcome this problem is to transfer compressed version of the file. Since, compression of the file reduces the size of file.

We have implemented a multithreaded self-adaptive file transfer client system, which uploads files on server over internet. The client system uploads a file on the server, but when bandwidth is found below a known threshold value, a component is dynamically loaded from the 'Repository' which compresses the file and uploads the compressed file on the server. The system can be categorized in three segments. These are Execution Monitor, Extractor and Repository.

Execution Monitor. The segment consists of mainly four components which are 'Client', 'Monitor', 'NormalUploader' and 'BandwidthChecker'. The Client is a

Graphical User Interface (GUI). The Monitor automates the self-adaptive system. Whenever a failure occurs in the system, the Monitor interacts with the Extractor and integrates a new component from the Repository, in order to assure normal behavior. The NormalUploader and BandwidthChecker components execute concurrently in different threads. The NormalUploader uploads a file to server over internet and the BandwidthChecker assembles, evaluates and examines bandwidth available to the client computer. If bandwidth is encountered below a known threshold value, BandwidthChecker stops normal file transfer and generates system failure event.

Extractor. The segment consists of only one component i.e. Extractor. It searches components in the Repository, on system failure. To fetch best suitable component available from the Repository, we use the Abstract Syntax Tree (AST) based approach as explained in previous work [10].

Repository. The segment consists of large number of components. For example it contains a ZIPP component. The system will adapt ZIPP component when NormalUploader component has been stopped due to bandwidth below known threshold value. The ZIPP component compressed the normal file and uploads the compressed file on the server. So, when client computer suffer from bandwidth below threshold, the Monitor integrates the ZIPP component from Repository with the help of the Extractor component.

5.2 The Custom CLR Profiler

One means by which we can observe the .NET runtime in action is by using the profiling API provided by .NET Framework [11]. The profiling API consists of COM interfaces such as callback interfaces (ICorProfilerCallback [12] & ICorProfilerCallback2 [13]) and info interfaces (ICorProfilerInfo [14] & ICorProfilerInfo2 [15]). To profile .NET 2.0 or later applications, implementation of callback interfaces are required; the info interfaces can be used to get information while inside event callbacks.

We have designed ‘CProfiler’ class which implements ‘ICorProfilerCallback2’ interface and maintains ‘CComQIPtr’ smart pointers [16] to info interfaces. The following code snippet represents code for the same in VC++ language.

```
class CProfiler : public ICorProfilerCallback2
{
  CComQIPtr<ICorProfilerInfo> pICPInfo;
  CComQIPtr<ICorProfilerInfo2> pICPInfo2;
…}
```

Initialize and Shutdown events. The CLR raised ‘Initialize’ and ‘Shutdown’ events of ICorProfilerCallback when test application is started to execute and terminate, respectively. In ‘Initialize’ event handler, registration of events (such as Assembly load/unload, Thread created/destroyed) are done by calling SetEventMask method;

and hooks are set for Function Enter/Leave by calling SetEnterLeaveFunctionHooks2 methods.

The 'Shutdown' event handler will generate trace file. The trace file is useful to generate trace diagram.

AssemblyLoadFinished event. CLR raises the 'AssemblyLoadFinished' event when an assembly load is finished by CLR and passes its AssemblyID as parameters. GetAssemblyInfo method can be used to get its name. The following code snippet (in Visual C++) depicts the filtration of Utility Assemblies.

```
pICPInfo->GetAssemblyInfo(aId,BUFSIZE,&dummy,name,&adId,&mId);
aMap[aId] = FilteredAssembly(W2A(name));
```

In above code snippet 'FilteredAssembly' returns true if assembly is not system utility, otherwise false.

Function Enter/Leave hooks and Filtration. CLR raises events on function enter and leave, which is handled by 'FunctionEnter' and 'FunctionLeave' hooks. The 'GetCurrentThreadID' method provides the 'ThreadID' of current thread. The 'GetTokenAndMetaDataFromFunction' method gives required information of function. So, using assembly and class name profiler will filter out utility class methods and further intra-component interactions;

Thread Created/Destroyed. CLR raises 'ThreadCreated' and 'ThreadDestroyed' events when a thread is created and destroyed, respectively; which passes the 'ThreadID' as argument. The profiler maintains separate method stack for each thread. So, the new stack allocated in 'ThreadCreated'.

5.3 Profiler Launcher

The CLR is the engine that drives .NET applications. When the CLR begins a process, it looks for two environment variables, COR_ENABLE_PROFILING and COR_PROFILER. If COR_ENABLE_PROFILING variable is set to 1, then CLR should use a profiler. Since profilers are implemented as COM objects, COR_PROFILER variable will be set to the GUID of the 'CProfiler' class. The following code snippet (in C#) depicts the Main method of Profiler Launcher application.

```
ProcessStartInfo psi = new ProcessStartInfo (exeFilePath);
psi.EnvironmentVariables.Add("COR_ENABLE_PROFILING", "1");
psi.EnvironmentVariables.Add("COR_PROFILER", PROFILER_GUID);
Process.Start(psi);
```

The next section shows the Trace File generated as a result of customized profiling.

5.4 Result: Trace File of Scenario

The trace file entries given below can be interpreted under following headers.
TimeStamp : Event ClassName :: MethodName : ThreadID

```
0: Initialize
3778285414: Enter Client :: Main : 5967392
12949253809: Enter Client :: browseButton_Click : 5967392
17998621752: Leave Client :: browseButton_Click : 5967392
20354749165: Enter Client :: uploadButton_Click : 5967392
20430626465: Enter Monitor :: startMonitor : 118454736
20614593492: Leave Client :: uploadButton_Click : 5967392
20717467072: Enter BandwidthChecker :: StartCheck : 130654872
21044261842: Enter NormalUploader :: Upload : 118454736
21693624120: Enter BandwidthChecker :: IsBelowThreshold : 118454736
21693729410: Leave BandwidthChecker :: IsBelowThreshold : 118454736
22497388195: Enter BandwidthChecker :: IsBelowThreshold : 118454736
22497510535: Leave BandwidthChecker :: IsBelowThreshold : 118454736
22497576697: Leave NormalUploader :: Upload : 118454736
22563788473: Enter Extractor :: Extract : 118454736
22736080821: Leave BandwidthChecker :: StartCheck : 130654872
24171053766: Enter Repository.Zipp :: Upload : 118454736
24171151952: Leave Repository.Zipp :: Upload : 118454736
24171232190: Leave Extractor :: Extract : 118454736
24171293043: Leave Monitor :: startMonitor : 118454736
24190385691: Enter Client :: Dispose : 5967392
24358378565: Leave Client :: Dispose : 5967392
24374318829: Leave Client :: Main : 5967392
24418540126: Shutdown
Thread :: 5967392
409937: Thread Created
Thread :: 118454736
20365516900: Thread Created
24242300496: Thread Destroyed
Thread :: 130654872
20716281709: Thread Created
22736347729: Thread Destroyed
```

5.5 Interpretation: Trace Diagram of Scenario

Fig 4 shows the trace diagram of scenario, where first the ‘User’ requests for uploading a file using ‘uploadButton_Click’ event to the ‘Client’. The ‘Client’ passes this request to the ‘Monitor’ using ‘startMonitor’ method call in another thread than the normal UI thread. Then, ‘Monitor’ invokes two methods concurrently in different threads, one for uploading the file and another thread to check if bandwidth below threshold. The diagram depicts that the normal file transfer fails before its completion because the bandwidth was found below the threshold value, so, the system adapts itself and switches to ‘Zipp’ after extracting it from ‘Repository’.

Note: The timestamp used in the trace diagram is relative to ‘Initialize’ event. Therefore, no unit of timestamp mentioned in the trace diagram.

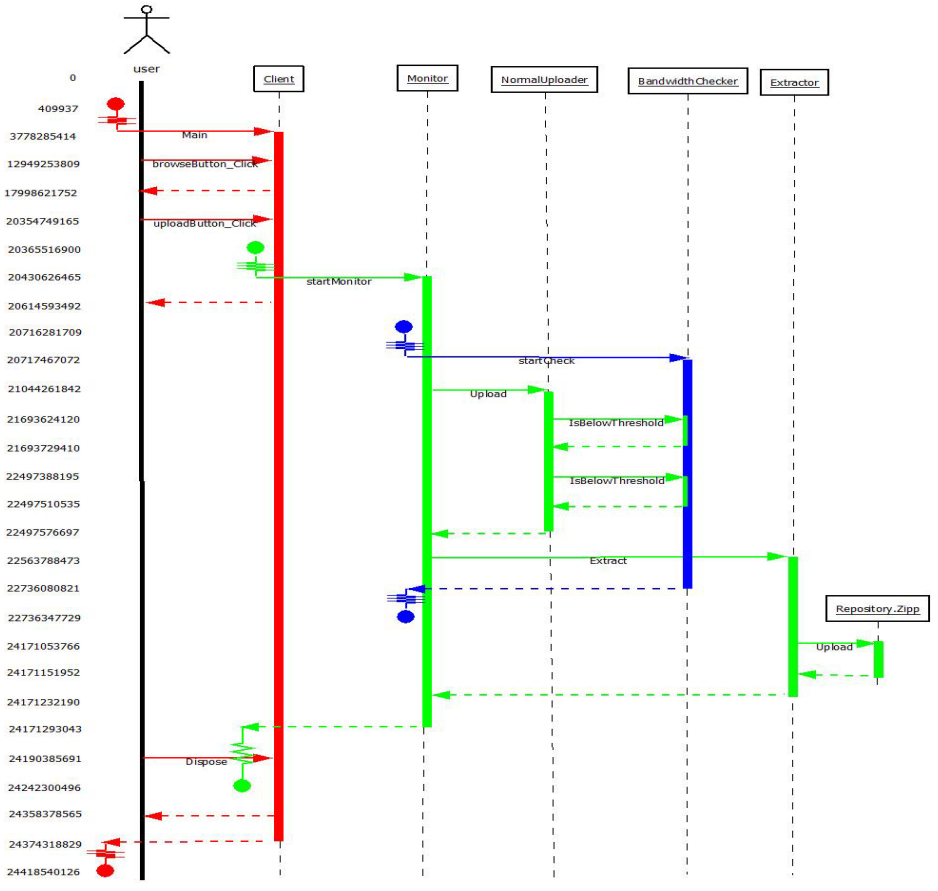

Fig . 4. Trace Diagram of Scenario

6 Conclusion and Future Work

We have presented and implemented an approach in that we have merged the strengths of system's instrumentation with their execution, to specify the complex concurrent nature of software. Moreover, our work defined the approach that collects the execution flows of concurrent threads in trace file. Trace file have been used to casually describe the scenario of execution of software. In this paper we have proposed the approach that visualized the execution of concurrent threads. This can show appealing opportunity, particularly for the case of on-line validation of automated software.

By instrument the system at the middleware level, we could assess the parallelism of threads execution in complex software. Their feedback in terms of trace file is extremely useful in order to analyze complex nature of the software.

In future we will re-engineer the visualization software in order to increase the degree of usability and customization for the end-user. Further future work will include building the framework for automatic translation of trace file into formal modeling that may sound the possibility to include building the technique for the on-line verification of complex nature systems.

References

1. Garlan, D., et al.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer 37(10), 46–54 (2004)
2. Oreizy, P., et al.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems 14(3), 54–62 (1999)
3. An Approach for Tracing and Understanding Asynchronous Systems. ISR Tech. Report UCI-ISR-02-7 (2002)
4. Rountev, A., Kagan, S., Sawin, J.: Coverage criteria for testing of object interactions in sequence diagrams. In: Fundamental Approaches to Software Engineering (2005)
5. Kramer, J., et al.: Detecting Implied Scenarios from Execution Traces. In: WCRE, pp. 50–59 (2007)
6. Rountev, A., Kagan, S., Gibas, M.: Static and dynamic analysis of call chains in Java. In: International Symposium on Software Testing and Analysis, pp. 1–11
7. Malnati, G., Cuva, C.M., Barberis, C.: JThreadSpy: teaching multithreading programming by analyzing execution traces. In: International Symposium on Software Testing and Analysis, pp. 3–13 (2007)
8. Bertolino, A., Muccini, H., Polini, A.: Architectural Verification of Black box Component Based Systems. In: Proceedings of International Workshop on Rapid Integration of Software Engineering Techniques, Switzerland (September 13-15, 2006)
9. Under the Hood: The. NET Profiling API and the DNProfiler Tool, `http://msdn.microsoft.com/en-us/magazine/cc301725.aspx` (accessed November 30, 2009)
10. Mishra, A., Mishra, A.K.: Component assessment and proactive model for support of dynamic integration in self-adaptive system. ACM SIGSOFT Software Engineering 39(4), 1–9 (2009)
11. Profiling (Unmanaged API Reference), `http://msdn.microsoft.com/en-us/library/ms404386.aspx` (accessed November 29, 2009)
12. ICorProfilerCallback Interface, `http://msdn.microsoft.com/en-us/library/ms230818.aspx` (accessed December 2, 2009)
13. ICorProfilerCallback2 Interface, `http://msdn.microsoft.com/en-us/library/ms230825.aspx` (accessed December 2, 2009)
14. ICorProfilerInfo Interface, `http://msdn.microsoft.com/en-us/library/ms233177.aspx` (accessed December 2, 2009)
15. ICorProfilerInfo2 Interface, `http://msdn.microsoft.com/en-us/library/ms231876.aspx` (accessed December 2, 2009)
16. CComQIPtr class, `http://msdn.microsoft.com/enus/library/wc177dxVS.80.aspx` (accessed December 5, 2009)