AN EVALUATION OF CHESS AS A TOOL FOR TESTING APPLICATIONS INVOLVING CONCURRENCY

Pradeep Kumar T
Ashit Shakrani
CONTENTS

• Introduction 3

• About CHESS 4

• Concurrency bugs that CHESS can detect
  Deadlock
  Livelock
  DataRace

• CHESS Internals 4
  How does CHESS work?

• CHESS Tools 4
  CHESSBoard
  MCHESS
  Concurrency Explorer
  VSTS integration
  Managed CHESS API

• Problem Scenarios 6
  Running a simple test scenario
  Observation
  Scenario illustrating deadlock detection
  Observation

• CHESS in Real-world applications using SharpDevelop 9
  SharpDevelop
  Testing SharpDevelop class (Positive test)
  Observation
  How to run the CHESS test bed

• Steps 13
  Compile Solution
  Execute MCHESS

• Known Limitations 14
The purpose of this paper is to analyze CHESS, a testing framework for concurrency-based applications. It is used to identify Heisenbugs, which are bugs in concurrent applications that appear to be random, making them difficult to reproduce and fix. The paper explores and discusses the various applicable scenarios for CHESS and presents observations of its usefulness for several different scenarios.
ABOUT CHESS

CHESS is a set of tools created by Microsoft researchers to find subtle concurrency errors in multithreaded single-process Windows and .NET programs. CHESS is specifically designed for concurrency unit testing and requires a set of test functions, each testing a particular concurrency scenario in the program. CHESS exhaustively enumerates all thread schedules of a test function by systematically inserting preemptions at various points in a program’s execution.

Concurrency Bugs that CHESS can Detect

CHESS is useful for detecting several types of concurrency bugs, including:

**Deadlock**—Refers to a specific condition when two or more processes are each waiting for each other to release a resource, or more than two processes are waiting for resources in a circular chain.

**Livelock**—Similar to a deadlock, except that the states of the processes involved in the livelock constantly changes with regard to one another, with none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing. Livelock can sometimes occur when an algorithm is put in place to detect a deadlock and when attempting to recover simultaneously.

**DataRace**—When separate processes or threads of execution depend on some shared state and the critical atomic sections of the code are not marked mutually exclusive. DataRace detection is time consuming and has to be explicitly requested from CHESS.

CHESS Internals

**How does CHESS work?**
CHESS runs its own scheduler, systematically causing preemptions in the various threads of the code under test. It can record the state of the program and monitor the program trace for potential bugs. CHESS is based on an exploratory design whereby it starts by forcing a simple set of inter-leavings and progressively varies the inter-leavings until it detects a bug in the code. Once a bug is detected, CHESS can write out the state and the exact set of inter-leavings that it produced, so the bug can be replicated in the future without exploration.

CHESS can detect calls to the various threading application programming interface (API) primitives in both managed and unmanaged code. Note that CHESS only works for multithreaded single-process native applications.

**CHESS Tools**
CHESS can be utilized in three ways. Each approach addresses specific scenarios.

1. As a test host ([HostType("CHESS")]) for Visual Studio Team System 2008 that runs managed unit tests under the control of CHESS. This is convenient as the code under test can be easily navigated by the errors detected by CHESS during its explorations.
2. As a command line tool (mCHESS.exe) for managed .NET programs. In this instance, it can be used or running as part of continuous builds.
3. As a command line tool (wCHESS.exe) for unmanaged Win32 programs. It can be used for scenarios in which you have unmanaged components.

**CHESSBoard**
CHESSBoard is an interactive GUI application that simplifies typical user interactions with CHESS, such as launching CHESS runs and managing test results. Its main purposes include the following:
CHESS runs its own scheduler, systematically causing preemptions in the various threads of the code under test.
Problem Scenarios

A simple test scenario
The following is a simple implementation of a bank class where the access to the instance variable is not synchronized, leading to potentially incorrect operations. To reproduce the bug, a simple test is run in which a deposit and withdraw of the same amount are completed on different threads. The test asserts that the balance remains unchanged. In a non-concurrent scenario, this code works fine. However, we must prove it works in a concurrent scenario.

SIMPLE BANK IMPLEMENTATION:

```java
public class Account
{
    int balance;
    
    public Account(int initial_balance)
    {
        this.balance = initial_balance;
    }
    
    public void Deposit(int amount)
    {
        this.balance += amount;
    }
    
    public void Withdraw(int amount)
    {
        this.balance -= amount;
    }
    
    public int GetBalance()
    {
        return balance;
    }
}

//TEST1
public class Test1
{
    public static bool Run()
    {
        Account account = new Account(10);

        // set up two threads to withdraw/deposit concurrently
        Thread t = new Thread(() => account.Withdraw(5));
        Thread s = new Thread(() => account.Deposit(5));

        t.Start();
        s.Start();
        t.Join();
        s.Join();
    }
}
```
return (account.GetBalance() == 10);
}

//TEST2
public class Test2
{
    public static bool Run()
    {
        Account account = new Account(10);
        int bal = 0;

        // set up 3 threads to withdraw/deposit/
        // getbalance concurrently
        Thread t = new Thread(() => account.Withdraw(5));
        Thread s = new Thread(() => account.Deposit(5));
        Thread r = new Thread(() => bal = account.GetBalance());
        t.Start();
        s.Start();
        r.Start();
        t.Join();
        s.Join();
        r.Join();
        return (account.GetBalance() == 10
                && (bal == 5 || bal == 10 || bal == 15));
    }
}

Observation
In a concurrent scenario, the balance variable is modified incorrectly. CHESS is able to reproduce the issue quickly by detecting the number of threads in the test (in this case two) and executing them with inter-leavings (preempting and resuming) in various patterns and in such a way that the balance variable invariably gets set incorrectly. Once the test fails, CHESS writes out the trace fail [debug trace] which shows the exact set of steps the threads went through when the test failed. The traces can be viewed with the concurrency tool. The test can then be fixed by introducing a lock over the access of the balance variable in a simple naive fix.

Scenario illustrating deadlock detection
This scenario illustrates the use of CHESS hosted in a Visual Studio environment.
DEADLOCK SAMPLE:

```csharp
//TEST
[TestClass]
public class Deadlock
{
    [TestMethod]
    [HostType("Chess")]
    [TestProperty("ChessExpectedResult", "Deadlock")]
    [TestProperty("ChessCheckRepro", "1")]
    public void DeadlockTest()
    {
        Test9.ChessTest.Main(null);
    }
}

//PROGRAM
public static bool Run()
{
    Thread t = new Thread(Child);
    t.Start();
    Parent();
    t.Join();
    return true;
}

public static void Parent()
{
    lock (lock1)
    {
        lock (lock2)
        {
            x++;
        }
    }
}

public static void Child()
{
    lock (lock2)
    {
        lock (lock1)
        {
            x++;
        }
    }
}
```

**Observation**
CHESS is able to keep track of the synchronization primitives, heuristically deduce the presence of lock and report it as such.
CHESS IN REAL-WORLD APPLICATIONS USING SHARPDEVELOP

SharpDevelop

SharpDevelop is an open source integrated development environment (IDE) for .NET languages (C#, F#, etc.). Analyzing the source code reveals the use of threading for a service locator pattern and for synchronized access in a console class. We use CHESS to investigate whether the code has any concurrency issues and present our observations.

Testing SharpDevelop class (Positive test)

```
SHARPDEVELOP:
   public class ThreadSafeServiceContainer : IServiceProvider, IDisposable
   {
      Dictionary<Type, object> services = new Dictionary<Type, object>();

      public ThreadSafeServiceContainer()
      {
         services.Add(typeof(ThreadSafeServiceContainer), this);
      }

      public object GetOrCreateService(Type type, Func<object> serviceCreator)
      {
         lock (services) {
            object instance;
            if (!services.TryGetValue(type, out instance)) {
               instance = serviceCreator();
               services.Add(type, instance);
            }
            return instance;
         }
      }

      public void TryAddService(Type type, object instance)
      {
         lock (services) {
            if (!services.ContainsKey(type))
               services.Add(type, instance);
         }
      }

      public object GetService(Type type)
      {
         lock (services) {
            
```
```
object instance;
    if (services.TryGetValue(type, out instance))
        return instance;
    else
        return null;
}

public void Dispose()
{
    IDisposable[] disposables;
    lock (services) {
        disposables = services.Values
            .OfType<IDisposable>().ToArray();
        services.Clear();
    }
    foreach (IDisposable disposable in disposables)
        disposable.Dispose();
}

public class ThreadSafeScriptingConsole :
    IScriptingConsole, IDisposable
{
    IScriptingConsole
        nonThreadSafeScriptingConsole;
    IControlDispatcher dispatcher;
    ThreadSafeScriptingConsoleEvents consoleEvents;

    delegate string ThreadSafeReadLineInvoker(int autoIndentSize);
    delegate string
        ThreadSafeReadFirstUnreadLineInvoker();

    public ThreadSafeScriptingConsole(IScriptingConsole nonThreadSafeScriptingConsole, IControlDispatcher dispatcher)
        : this(nonThreadSafeScriptingConsole, new ThreadSafeScriptingConsoleEvents(), dispatcher)
    {
    }

    public ThreadSafeScriptingConsole(IScriptingConsole nonThreadSafeScriptingConsole,
        ThreadSafeScriptingConsoleEvents consoleEvents, IControlDispatcher dispatcher)
    {
        this.nonThreadSafeScriptingConsole = nonThreadSafeScriptingConsole;
        this.consoleEvents = consoleEvents;
        this.dispatcher = dispatcher;

        nonThreadSafeScriptingConsole.LineReceived += NonThreadSafeScriptingConsoleLineReceived;
    }
void NonThreadSafeScriptingConsoleLineReceived(object source, EventArgs e)
{
    consoleEvents.SetLineReceivedEvent();
}

public void WriteLine()
{
    if (dispatcher.CheckAccess()) {
        nonThreadSafeScriptingConsole.WriteLine();
    } else {
        Action action = WriteLine;
        dispatcher.Invoke(action);
    }
}

public void WriteLine(string text, ScriptingStyle style)
{
    if (dispatcher.CheckAccess()) {
        nonThreadSafeScriptingConsole.WriteLine(text, style);
    } else {
        Action<string, ScriptingStyle> action = WriteLine;
        dispatcher.Invoke(action, text, style);
    }
}

public void Write(string text, ScriptingStyle style)
{
    if (dispatcher.CheckAccess()) {
        nonThreadSafeScriptingConsole.Write(text, style);
    } else {
        Action<string, ScriptingStyle> action = Write;
        dispatcher.Invoke(action, text, style);
    }
}

public string ReadLine(int autoIndentSize)
{
    consoleEvents.ResetLineReceivedEvent();
    string line = ThreadSafeReadLine(autoIndentSize);
    if (line != null) {
        return line;
    }
    if (consoleEvents.WaitForLine()) {
        return ThreadSafeReadFirstUnreadLine();
    }
    return null;
}
string ThreadSafeReadLine(int autoIndentSize)
{
    if (dispatcher.CheckAccess()) {
        return nonThreadSafeScriptingConsole.
            ReadLine(autoIndentSize);
    } else {
        ThreadSafeReadLineInvoker action =
            ThreadSafeReadLine;
        return (string)dispatcher.Invoke(action,
            autoIndentSize);
    }
}

string ThreadSafeReadFirstUnreadLine()
{
    if (dispatcher.CheckAccess()) {
        return nonThreadSafeScriptingConsole.
            ReadFirstUnreadLine();
    } else {
        ThreadSafeReadFirstUnreadLineInvoker
            action = ThreadSafeReadFirstUnreadLine;
        return (string)dispatcher.Invoke(action);
    }
}

public void Dispose()
{
    nonThreadSafeScriptingConsole.LineReceived -=
        NonThreadSafeScriptingConsoleLineReceived;
    consoleEvents.SetDisposedEvent();
}

public event EventHandler LineReceived;
protected virtual void OnLineReceived(EventArgs e)
{
    if (LineReceived !$= null) {
        LineReceived(this, e);
    }
}

public void SendLine(string line)
{
    nonThreadSafeScriptingConsole.
        SendLine(line);
}

public void SendText(string text)
{
    nonThreadSafeScriptingConsole.
        SendText(text);
}

public string ReadFirstUnreadLine()
{
    return nonThreadSafeScriptingConsole.
        ReadFirstUnreadLine();
}
**Observation**

Going through the code, we found no concurrency issues and CHESS returned the same verdict. CHESS increased the number of threads and exploration patterns until it exhausted all available patterns trying to find any bugs.

Note that the preemption bounds can be increased manually.

How to run the CHESS test bed

The steps for conducting the above test against SharpDevelop are stored in the SDTest folder:

- Class under test - ThreadSafeServiceProvider.cs
- TestClass - ThreadSafeServiceProviderTest.cs - contains a single test
- testlist.xml - the input for mCHESS, a file describing the tests to be run and the solution containing the test file

**STEPS**

1. **Compile Solution**

    ```
c:\Windows\Microsoft.NET\Framework\v3.5\csc.exe /target:library ThreadSafeServiceProvider.cs ThreadSafeServiceProviderTest.cs /reference:nunit.framework.dll
    ```

2. **Execute mCHESS**

    ```
    "c:\Program Files (x86)\Microsoft CHESS\bin\mchess.exe" /outputprefix:TSP ThreadSafeServiceProvider.dll /showprogress echo %errorlevel%
    ```
Known Limitations

As mentioned earlier, CHESS is limited to discovering bugs in multi-threaded, single-process applications. For programs that have large sections of code with several threads running at the same time, the amount of explorations that CHESS must do increases exponentially, causing tests to take longer than anticipated.

CHESS tools have been created for .NET framework 3.5 and vs2008, but are not currently updated to target .NET framework 4 assemblies and don’t integrate with vs2010.

CHESS tools have minor bugs in them hampering their effective usage. For example, the exit codes for mCHESS were sometimes incorrect and didn’t follow the pattern as mentioned in the mCHESS doc. In addition, mCHESS didn’t always give an indication of what went wrong when it failed to produce traces.

Suggested improvements to CHESS include:

- Enhancing the exploratory design to use previous traces to test high possibility error zones and reduce running time.
- Better integration of ChessBoard within Visual Studio in order to see CHESS output.
- Fixing minor bugs in mCHESS and providing support for .NET 4.

Conclusion

In this paper, we evaluated CHESS to determine whether it can be used to identify difficult-to-reproduce bugs in concurrent applications. We analyzed the scenarios that CHESS can handle and discussed how to set up a test scenario with a real-world application, gathering our observations of CHESS output.

We can conclude from our analysis that CHESS is useful as it brings in a framework that can be used to test and confidently reproduce previously unverifiable bugs in concurrent applications. We found that CHESS does have a few limitations, however. The fact that the exploratory design will scale with a larger codebase and the lack of update and minor bugs in the CHESS toolkit are of concern.

[1] Temporarily interrupting a task being carried out by a computer system without requiring its cooperation, and with the intention of resuming the task at a later time.

[2] Walking through a state and time space with depth and with various branches until a specified precondition is met.

© 2011
Global Offices

**Bangalore**
Salarpuria GR Tech Park
6th Floor, “VAYU” Block
#137, Bengaluru 560066
Karnataka
India
Tel: +91 (080) 410 47 000

**Boston**
131 Dartmouth Street
3rd Floor
Boston, MA 02116
Tel: +1 (617) 621 0200

**Calgary**
888 3rd Street SW
Suite 1000
Calgary, Alberta T2P 5C5
CANADA
Tel: +1 (403) 444 5574

**Chicago**
30 West Monroe,
12th floor
Chicago, IL 60603
Tel: +1 (312) 458 1800

**Delhi**
Towers D & E,
DLF Cyber Greens
DLF Phase III
Sector 25-A
Gurgaon 122002
Haryana
India
Tel: +91 (124) 416 7000

**Dusseldorf**
Hammer Straße 19
40219 Düsseldorf
Germany
Tel: +49 (0) 211 540 34 0

**Houston**
Heritage Plaza
1111 Bagby Street Suite
1950
Houston, TX 77002
Tel: +1 (713) 493 6880

**London**
Eden House
8 Spital Square
London, E1 6DU
UNITED KINGDOM
Tel: +44 (0) 207 786 4500

**Los Angeles**
1601 Cloverfield Blvd.
Suite 400 South
Santa Monica, CA 90404
Tel: +1 (310) 264 6900

**New York**
One Penn Plaza
24th Floor, Suite 2400
New York, NY 10119
Tel: +1 (212) 560 5700

**Singapore**
Air View Building #02-01
2 Peck Seah Street
Singapore 079305
Republic of Singapore
Tel: +65 (3) 104 5308

**Toronto**
129 Spadina Avenue
Suite 500
Toronto, Ontario M5V 2L3
CANADA
Tel: +1 (416) 645 1500

**Zurich**
Sapient Switzerland GmbH
Seefeldstrasse 35
8008 Zurich
Switzerland
Tel: +41 (58) 206 06 00