Installing and Running ManagedCHESS

# Introduction

ManagedCHESS is a tool for finding subtle errors due to concurrency in a multithreaded .NET program. ManagedCHESS exhaustively enumerates all thread schedules in such a program by systematically inserting context-switches at various points in the program. Moreover, if a particular thread schedule results in an error, ManagedCHESS helps in reproducing the error by reproducing the error-causing schedule.

Other instances of CHESS include Win32CHESS for user-level Win32 applications and SingularityCHESS for the Singularity operating system.

See <http://codebox/chess/> and <http://research.microsoft.com/projects/chess/> for more details.

# Installing ManagedCHESS

Create a new enlistment of the CHESS source code from <http://codebox/chess/SourceControl>. In the main directory run “sd sync” to get the latest/greatest CHESS bits. Use VS2008 to open ManagedCHESS.sln and compile CHESS. If you are using Vista, remember to open VS2008 with administrative privileges. You can use the debug or the release build configuration. All CHESS executables are deposited in main\debug[release].

Running ManagedCHESS

ManagedCHESS currently works for user-mode applications that reside in a single process. It takes as input a test executable (Test.exe) that uses the System.Threading API for creating threads and synchronizing among them. In order to be analyzed by ManagedChess, Test.exe must contain a public class named ChessTest containing:

* public static bool Startup(string [] args) { … } *// optional*
* public static bool Run() { … } *// required*
* public static void Shutdown() { … }  *// optional*

The role of these methods is explained below. For a complete list of the supported interfaces see the section “Requirements for ChessTest.Run”. Run Test.exe under the control of ManagedChess via the command

managedchess.exe [/csb:<n>] Test.exe [/arg:<arg1> /arg:<arg2> ... /arg:<argn>]

Note that the main routine of Test.exe is not executed when Test.exe is run under the control of ManagedCHESS. In this way, you can use the same test executable for your regular testing and ManagedCHESS testing.

ManagedCHESS executes "ChessTest.Startup" before the start of the test, then runs the test function ChessTest.Run repeatedly such that each iteration takes a different thread schedule, and then executes "ChessTest.Shutdown" after all the schedules are exhausted. In other words, ManagedCHESS executes the following pseudo-code:

ManagedChess.Main(string[] args){

if (ChessTest.Startup(args)) { // if startup succeeds,

while(!ChessDone()) { // as long there’s another schedule

if (!ChessTest.Run()) { // run the test

ChestTest.Shutdown(); // the test failed

Exit(-1)

}

} // the test succeeded, iterate

ChestTest.Shutdown(); // all schedules investigated

Exit(0); // with no error

} else {

Exit(1); // startup failed

}

}

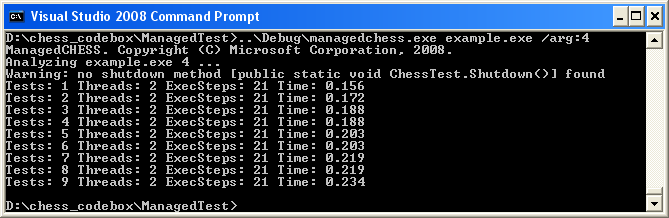
Users can use the "ChessTest.Startup" function to set up the test environment. The arguments to Test.exe (passed via /arg) are provided (as usual in .NET) via an array of strings to ChessTest.Startup. This function should return “true” if the startup succeeded. If anything went wrong during startup, the function should return “false”. The test function “ChessTest.Run” should return true if the test succeeded and false if it failed. If the function throws an exception, ManagedCHESS will catch it and count this as a failure of the function.

# Example

Let’s run ManagedCHESS on the program example.cs in the ManagedTests directory. First, compile the file via “csc /debug example.cs”. Now run ManagedCHESS as follows

..\Debug\managedchess.exe example.exe /arg:4

When you run ManagedCHESS, you will see output like the following:

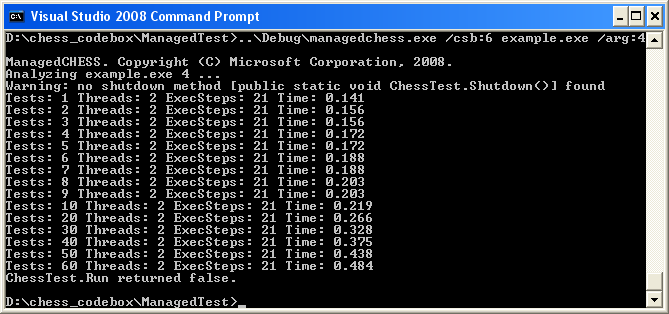


In this output, the last line indicates that ManagedCHESS has executed 9 different thread schedules, the maximum number of threads in any schedule so far is 2, the maximum length (measured in the number of synchronization operations) of any schedule so far is 21, and 0.265 seconds have elapsed since ManagedCHESS started.

You can control the search performed by ManagedCHESS using the option "/csb" that stands for context-switch-bound. By passing "/csb:n" to ManagedCHESS, you instruct it to explore thread schedules with at most n preemptions. The number of executions explored by ManagedCHESS grows exponentially with the bound. We have found small bounds such 0, 1, or 2 to be good enough to expose many concurrency errors. By default, csb is set to 1. However, we have constructed example.cs so that it requires at least 6 preemptions to cause ChessTest.Run to fail. Let’s run again with such a context bound:

..\Debug\managedchess.exe /csb:6 example.exe /arg:4

Which results in

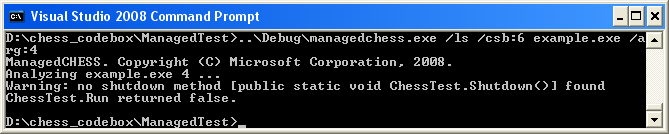


If ManagedCHESS executes a schedule that causes an error, the error will manifest itself exactly as it would if you executed your test program without CHESS.

In addition to finding concurrency errors, CHESS also solves the problem of reproducing them. Before CHESS executes a thread schedule, it writes out an encoding of it on the disk. If the program crashes, you can reproduce the error by running CHESS again with the "/ls" option. Let’s look at the schedule that caused the example program’s Run method to fail:

..\Debug\managedchess.exe /ls /csb:6 example.exe /arg:4

Now CHESS replays the last thread schedule without performing a search, resulting in the following:



You can thus make the program repeat the error as often as you want and debug the problem with the aid of a debugger.

# Guidelines for ChessTest.Run

If the test function ChessTest.Run does not satisfy the following conditions, you might run into the problems described in the section "Problems".

1. ChessTest.Run should be idempotent. In other words, any complete execution of the function should leave the program state behaviorally equivalent to the state it started in. This requirement has many implications:
   * ChessTest.Run should free all resources acquired by it including memory and kernel handles. In particular, it must wait for the termination of any threads it creates. If the function uses a thread pool, then by the use of appropriate signaling it must ensure that any work items initiated by it have finished.
   * ChessTest.Run should reset any global variables (files, databases, etc.) that influence its behavior.
   * ChessTest.Run should reset any hardware state that influences its behavior including files, memory buffers, etc.
2. ManagedCHESS currently supports most of System.Threading. If you use concurrency primitives outside this namespace, ManagedCHESS will not be able to control the execution of your program. Please let us know which concurrency APIs are important to you.
3. Test functions often use random numbers. We recommend that you instead use the function "MChessChess.Choose" exported by MChess.DLL (a managed wrapper around the unmanaged Chess.DLL)

requires n > 0

ensures 0 <= return < n

int Choose(int n)

MChessChess.Choose(n) will return an integer in the range [0,n). The advantage of using Choose rather than random numbers is that CHESS will systematically explore all the possible choices. Note that Choose should be used only to cover many control paths. It is better to use random numbers for creating random data values or memory buffers. If you do use random numbers, make sure that ChessTest.Run always starts with the same random seed.

1. Although ManagedCHESS will work with all test functions, it is best if ChessTest.Run creates no more than 2-5 threads and executes no more than 100-200 steps in any schedule. You can find out the number of threads and steps in your test by running ManagedCHESS and looking at the output generated.

# Helping CHESS help you

The CHESS scheduler systematically inserts context switches in your program. Thus, it is very good at creating 'stress' scenarios even with a few threads, events, or messages. The number of schedules explored by CHESS grows very fast as the number of threads and events in the test increases. Therefore, in order to help CHESS cover the state space of your program well, it is important to have enough threads and events in the program but just as important to have no more.

A good way to design a test case is to perform the following thought experiment. Think of the hardest concurrency bug in the system that you know of. Then think of the simplest explanation for the bug, in the number of threads, messages, and events etc. Then create a test program with those parameters and let CHESS run on it. Finally, write as many assertions in your program as possible.

Problems

1. **Failure due to nondeterministic tests:** CHESS is designed only to handle nondeterminism arising from concurrency and asynchrony in the input test case. This includes the execution order of threads, timers, and asynchronous callbacks from the kernel. For efficiency reasons, CHESS does not handle other sources of nondeterminism. Examples of such sources include using the current time of day, registry values, or files in the disk. When CHESS detects external nondeterminism, it stops the execution of the test and reports an error message. You can tell CHESS to continue search
2. **Max-execution timeout:** If the test case uses synchronization APIs outside those supported in CHESS, the execution typically leads to a deadlock resulting in an execution timeout (default set to 50 seconds).

# ManagedCHESS Samples

The directory ManagedTests contains a few sample tests that demonstrate the functionality of ManagedCHESS.

1. Open a Visual Studio 2008 Command Prompt. Alternately, you can run vcvars32.bat (usually installed in C:\Program Files\Microsoft Visual Studio 9\VC\bin) from any command prompt.
2. Each sample is in a .cs file. You can build the samples using nmake. For instance, to build example.cs, you should run 'nmake example.exe' in the ManagedTest directory. Run the example, as explained earlier in this document.
3. Run the test again increasing the context-switch bound, by specifying the /csb:6 argument. Now, ManagedCHESS explores more thread schedules due to the increased csb. (The previous execution ran with the default bound of 1.). After around 60 schedules, ManagedChess should stop with the message that “ChessTest.Run return false”.
4. If you run CHESS again, specifying the /ls and /gui arguments, you will see the error trace displayed in the Concurrency Explorer GUI.
5. There are other samples in the ManagedTest directory. You can try them out one by one as described above. Alternately, you can use our regression perl script to run them as follows:

nmake all

The output of the regression can be compared with expected output by using windiff

windiff out.\* golden.\*

# ManagedCHESS exit codes

ManagedChess.exe returns the following exit codes

|  |  |
| --- | --- |
| Error code | Reason |
| 0 | Normal exit |
| -1 | Test exited abnormally |
| -2 | CHESS detected a deadlock |
| -3 | CHESS detected a livelock |
| -4 | CHESS detected a timeout |
| -5 | CHESS detected nondeterminism outside its control |
| -6 | Test does not satisfy the requirements described above |

# Advanced CHESS options

### Volatile Variables

You can enable tracking of reads and writes to fields declared “volatile” with the /volatile option to ManagedCHESS. Please note that this option significantly slows down ManagedCHESS due to two reasons: (1) the overhead of tracking volatile access; (2) the increased number of schedules to explore.

### Inserting explicit context switches

By default, CHESS automatically inserts a context switch right before every synchronization function. Sometimes, you might want to force a context switch at other places in your code. You can do so by calling ChessSchedulePoint() (defined in Chess/ChessApi.h). Please use ChessSchedulePoint() sparingly.

### Excluding assemblies and types from analysis

By default, ManagedChess analyzes all assemblies that are dynamically loaded. You can tell ManagedChess to exclude an assembly from analysis via the option /ea:<assembly>. For more precision, you can tell ManagedChess to exclude all methods associated with a type from analysis via the option /et:<type>.

### Additional debugging help

You can use the following options to automatically break during CHESS execution

|  |  |
| --- | --- |
| Command line option | Break() at |
| /brk:s | Start of execution |
| /brk:a | Assertion failures |
| /brk:c | Context switches |
| /brk:p | Preemptions (context switches where the current thread is still enabled) |
| /brk:d | A deadlock |
| /brk:t | A timeout |

Note: to supply multiple /brk options (such as “s” and “a”), contenate the characters together into one string (/brk:sa).

### GUI

ManagedCHESS has a prototype GUI, still under development. You can invoke the GUI by providing the /gui option to ManagedChess.exe

# ManagedChess Contacts

Please send email to [chequers@microsoft.com](mailto:chequers@microsoft.com) if you run into any problems. We also welcome your feedback. For more information visit: <http://research.microsoft.com/chess/>. Also, you can subscribe to mailing lists chessann (CHESS announcements) and chessdis (CHESS discussions) using <http://autogroup>.

|  |
| --- |
| Thomas Ball ([tball@microsoft.com](mailto:tball@microsoft.com))  Madan Musuvathi ([madanm@microsoft.com](mailto:madanm@microsoft.com))  Shaz Qadeer ([qadeer@microsoft.com](mailto:qadeer@microsoft.com)) |