Tyche 3.0 Simulation Engine Project Review

Prepared By:
Terry Restoule
LeverageTek IT Solutions
136 Lewis Street Suite 1, Ottawa, ON, K2P 0S7
Contract Number: W7714-093810
Contract Scientific Authority: Cheryl Eisler, JTFP/MARPAC Team, 250-363-4830
Published in October, 2014 as DRDC-RDDC-2014-C220

Version 3.0
Latest Revision
October 9, 2014

DRDC-RDDC-2014-C220

RECORD OF AMENDMENTS

Amendment No.	Entered By	Revisions	Date
1	Terry Restoule	Original Version	Jun 18, 2014
2	Cheryl Eisler	Revisions	Jul 29, 2014
3	Cheryl Eisler	Abstract Revision	Oct 9, 2014

ABSTRACT

The Tyche Simulation Engine was first developed in Microsoft Visual Basic 6. With that platform becoming dated and with performance becoming an issue, it was decided that the engine should be migrated to a more contemporary platform to improve performance and to enable functional expansion. This contractor report documents the history of the redevelopment project, with a focus on improvements that were made following the translation of the code from Visual Basic 6 to Microsoft Visual C#.NET. This includes addressing known bugs, extensively restructuring the code to improve maintainability, as well as a number of new development initiatives that were then investigated for feasibility. Some, such as new random number generation and user selected timescales were implemented. Other development initiatives were investigated and either postponed to a future version of the software because of the amount of restructuring that would be required, or abandoned due to complexity of implementation.

TABLE OF CONTENTS

1. INTRODUCTION

Over the past three years, the Tyche Simulation Engine has been transitioned from Visual Basic 6 (no longer supported by Microsoft) to Visual C#.NET. Once redeveloped, performance improvements made possible several enhancements. Some of those enhancements were realized while others were investigated and either abandoned or postponed.

Before the engine could be redeveloped, several known problems with the Visual Basic version were addressed so that a reliable version would be available for use while the new version was being developed. An extensive evaluation of several potential development platforms was then undertaken and Visual C# emerged as the preferred development choice.

Along with the new programming language, the data structures used within the engine were transformed to take advantage of performance improvements seen during the platform evaluation.

The following sections summarize the work that has been done and what has been learned over the course of the project.

2. VISUAL BASIC VERSION CHANGES

Prior to redeveloping the simulation engine, several known inconsistencies/problems with the Visual Basic version were addressed. These included:

- Error handling was expanded, corrected and standardized across the engine.
- A problem with the version of WinZip being used to zip the simulation output was fixed.
- Problems with the code interfacing with macros in the Risk.XLS were fixed.
- The Base form in the GUI was not working properly when more than two bases were entered. This was fixed.
- Whenever a file was closed in the GUI the user would be asked to save changes even when no changes were made. This was fixed.
- There was an "Invalid Function Call" error that would occur approximately 100 iterations into a simulation. This was fixed.

3. REDEVELOPMENT SOFTWARE EVALUATION

Prior to redeveloping the Tyche software an evaluation[1,2] was conducted involving the following development platforms:

- Visual Basic.NET (VB.NET)
- Visual C#.NET (VC#)
- Visual C++.NET (VC++)
- Dev C++
- Python 2.7
- SimPy
- SimIO
- SimEvents (Matlab)

A set of required software features were identified, and a program template was developed to demonstrate those features. The template program was first written in VB6 to provide a performance benchmark.

Versions of the template program were then developed for each of the programming language platforms and simulation packages being evaluated. Strengths, weaknesses and problems were identified. Timing information was collected for each of the test programs over multiple iterations. Performance tests were conducted on a common computer.

All of the potential development languages generally out-performed Visual Basic 6 in their tests. Overall, Visual C#.NET ran the fastest and was selected as the new development platform.

[1] Eisler, C. (2012), Tyche 3.0 Development: Comparison of Development Environments for a Monte Carlo Discrete Event Simulation (MCDES), (DRDC TM 2012-231), Defence R&D Canada – CORA.

[2] Restoule, T. (2012), Notes on the SimulTest Programs Used in the Language Selection Process: Tyche 3.0 Simulation Engine Project Development, (DRDC CR 2012-092), Defence R&D Canada – CORA, LeverageTek IT Solutions, Ottawa, ON.

4. ENGINE REDEVELOPMENT USING VISUAL C#.NET

With Visual C#.NET selected, redevelopment of the Simulation Engine began.

Initially, new code was written to be as much a direct translation as possible in order to allow the new system to be easily tested in parallel with the original Visual Basic code. Most of the C# code methods retained the names and calling structures of their corresponding functions and subroutines in the VB6 version. Variables in the C# version were defined with the same names as the class object variables used locally within the Visual Basic code. Bugs in the Visual Basic code were found during the process. Since the output from the C# application had to be identical in order to verify processing accuracy, the C# code had to be "broken" to match the output of the old code. When these situations occurred, the correct code was left in as comments and further comments were added to the code so that the C# code could be easily fixed at a later date.

During the technical evaluation, it was found that the representation of internal data as class objects and collections had performance drawbacks and it was decided that in the new version, data would be stored in structures (structs), dynamic arrays, and lists as appropriate.

The new system was considered functional when it produced output identical to the old on a specific set of test cases.

Once the C# version was deemed functional, the following strategies were used to improve the performance of the new system:

1. Removal of extraneous table look-ups.
2. The replacement of temporary working variables with direct references and updates to the global arrays.
3. Improved code re-use through shorter methods, eliminating duplication and making future upgrades easier.

In addition, the calling of many methods was simplified through changes in variable scoping. Many variables that had been defined local to a method and passed to other methods through parameters were redefined as global within the class.

Though the new C# version of the simulation engine showed a significant improvement in processing speed over the Visual Basic-based one, the speed improvement was not as dramatic as the results in the technology evaluation predicted[3].

[3] Restoule, T. (2013), Notes on the Conversion of the Tyche Simulation Engine from Visual Basic 6.0 to Visual C#.NET: Tyche 3.0 Development Project, (DRDC CR 2013-156), Defence R&D Canada – CORA, LeverageTek IT Solutions, Ottawa, ON.

5. POST-REDEVELOPMENT ACTIVITIES

As mentioned in the preceding section, to insure that the output from the new C# version of the Engine would match the old output exactly, when bugs were found in the old code, the new version had to be "broken" to match. Once the redevelopment of the engine was complete, the code was then "corrected".

In addition to these fixes, the following problems were addressed.

5.1 *REMOVAL OF BASE DISTANCES*

In Tyche 2.3.4, transit from location to location was done using two lookup tables. The "Distance" table was used when travelling from a base to a theatre and the "BaseDistance" table was used when travelling between bases. The two tables had different structures. This distance concept was carried forward into the C# version of the Engine due to the direct translation nature of the redevelopment. Based on the idea that "a distance is a distance", the BaseDistance table was dropped. The distance values between bases were added to the Distance table, and appropriate code changes were made in the code that deals with the input (TYI) files and in the places where BaseDistance lookups occurred.

5.2 *INTEGRATION WITH THE DASHBOARD PROGRAM*

The Tyche Simulation Engine is designed to work hand in hand with the Tyche Dashboard program. When a simulation is started, keys are written to the Windows registry uniquely identifying the simulation. The Dashboard informs the simulation engine to proceed by writing a value to the registry. At the end of each iteration, the simulation engine updates the registry with progress information. When the system was redeveloped, some of this "handshaking" through the registry did not work and had to be corrected.

5.3 *PROBLEMS WITH ASSET BUMPING AND ESSENTIAL ASSETS*

Assets used within the model can have dependencies on other assets. If a dependency exists that is vital to assignment (e.g. a ship and its crew), the assets are said to be "essential" to each other. When an asset is "bumped" from an activity (rescheduled), the assets essential to it are supposed to be bumped with it (unless the dependent asset is also essential to other assets in theatre). After redevelopment, it was noticed that this was not happening and had not happened in the original version either. This problem was due to two situations:

1. Essential assets were not being associated correctly and consistently
2. The "EssentialTo" list was not being navigated correctly when the asset was being bumped.

Once these problems were corrected, issues arising with follow-on events also had to be addressed.

6. IMPLEMENTATION OF USER SPECIFIED TIMESCALES

In the original Simulation Engine and in the redeveloped one, simulations ran over years with asset activities measured in days. Asset speeds were also fixed and based on distance travelled per day. There was no provision for a user to specify their own timescales.

To implement user specified timescales, the user would pick the scales they wanted to use in the GUI and their selections would then propagate from the TYI file to the Simulation Engine and then the output (TYO) file and run statistics.

Timescale fields were added to the beginning of the TYI file and the code within the engine that reads the TYI file was modified to read these new values. Backwards compatibility was maintained so that the Engine could still process the old format (defaulting to Years and Days). Fundamentally, the Engine code was changed so that all simulations ran in minutes (the smallest timescale). At the beginning of the model's processing, conversion constants were calculated allowing for conversion to the user's units from minutes and vice versa. All time related calculations and constants were adjusted using the conversion constants.

When an iteration's output was being written, time values were converted from minutes to the user's preferred units. Headings at the beginning of the TYO file were also adjusted to reflect the user's timescale selections. The user is provided with three possible options (Years/Days, Days/Minutes or Hours/Seconds).

These changes were successfully implemented.

7. OTHER DEVELOPMENT INITIATIVES

A number of other development initiatives were investigated. Some of these initiatives were pursued while others were dropped or postponed.

7.1 *NEW RANDOM NUMBER GENERATION*

High quality random number generation is essential to create useful and realistic simulations. Improvements to the Engine's random number generation were investigated and it was decided to replace the random number generator with one that was faster and had a longer period. In implementing the new generator, it was also decided to change the way in which random numbers were used.

In the original Engine, a single stream of random numbers was used to provide all of the random values needed throughout all of the iterations. This random sequence was seeded by an input parameter delivered through a command line value or XML file. Since the intention was to run Tyche in a multi-processor environment in the future, this approach would have been problematic, if not impossible, in such an environment.

It was decided to use two random streams. The first stream would be seeded with the input parameter and would generate seeds for each iteration of the model. The second stream would take that iteration seed and generate all of the random numbers needed within an iteration.

These changes were successfully implemented.

7.2 *THEATRE-TO-THEATRE TRANSIT*

Within the Tyche simulation, locations where activity can occur fall into 2 groups. "Bases" refer to DND installations that assets work from. "Theatres" are locations where scenarios occur. When assets move within the simulation they move from base to theatre and back. To more accurately reflect combat situations, assets should be able to transit directly between theatres as required. An assessment was made as to the impact of this transit change to the model.

The Tyche simulation is not geographically bound; event involvement, asset activity and transit are all measured in time. Theatre-to-Theatre transit involves viewing bases and theatres as just locations.

Along with several changes to the GUI, the simulation input file (.TYI) would need to be changed to replace the "Bases" and "Theatres" sections with a single "Locations" section and sets of travel times would need to be included for each Asset Type.

Inside the simulation engine, the Base and Theatre tables would be replaced by a Locations table and the SearchBases tables would be replaced by SearchLocations. The ComputeTransitTime method would lookup transit values based on the asset's type. Code changes would also need to be made to determine when an asset would return to base at the end of an event and when it would transit directly to the next event.

It was decided to postpone this enhancement, as the models would quickly grow unwieldy with this construct. For example, with just 10 locations (say 2 bases and 8 theatres previously that would require 17 pieces of information from the user) there would then be 45 pieces of information required. This would overwhelm the user for a part of the model that is often not of critical significance.

7.3 *MULTI-THREADING OF ITERATIONS*

The possibility of restructuring the simulation engine so that each iteration of a simulation could be executed individually on its own processor in a multi-processor environment using multi-threading was investigated. (Note that this is not full parallel processing in a high performance computing environment.)

A simulation would start normally: writing information to the registry, reading an input file, and setting up the simulation. When iterations began to be processed, each iteration's processing would be assigned to a thread running on one of the processors. In this way, several iterations could be processed at once (the number of simultaneous iterations being the number of available processors – 1, leaving one processor to monitor the overall process). When an iteration finished, another iteration would be threaded onto its processor, and this would continue until all iterations were completed. With the iterations done, the run statistics would be generated and the simulation would wrap up.

To investigate how threading might effect the simulation engine, a C# program was created that performed 1000 iterations of a process that created significant output. Each iteration a the test program opened and read an e-book and then printed it to a file. As a benchmark, it wrote the book to 1000 files in a non-threaded configuration and then it wrote the book 1000 times to a single text file in a non-threaded environment.

It was found that in the non-threaded situation, writing each iteration's output to its own individual file took substantially longer than writing all of the output to one file. This was also true when the books were written from individual threads. Writing all of the books to one file in a threaded situation was faster than writing to individual files, but not by much, because processing was slowed due to signaling delays (only one thread can write to the file at one time; if other threads need to write to the file, they must wait for the file to be relinquished).

There was also found to be significant overhead in creating threads, initiating processing on them and in disposing of them when they finished. To allow for these housekeeping chores, threads had to sleep for short periods but frequently.

In the end, though the multi-threaded processing of the test program was very slightly faster than the non-threaded processing, the improvement was slight enough that it might not even occur after restructuring the much more complex engine.

According to Microsoft, when .NET is running on a multi-processor (multi-core) computer, it uses all cores to optimize performance.

7.4 *HPC JOB SUBMISSION*

One of the objectives of the project was to have the simulation engine running on a computer running Windows HPC Server. The intention was to take advantage of the multi-processor environment available on an HPC cluster. Demonstration code was written toward the end of the project to show how the Tyche-SE software could be run through the HPC job scheduler and how progress information written to the registry over the course of the simulation run could be passed back to the scheduler through the Tyche-SE Job object.

8. CONCLUSION

The Tyche Simulation Engine software was converted from Microsoft Visual Basic 6 to Visual C#.NET. The software was directly translated to allow the new system to be easily tested in parallel with the original Visual Basic code. Once the new software produced output consistent with the Visual Basic version, several problems were addressed and the new code was extensively restructured to improve maintainability. A number of development initiatives were then investigated. Some, such as new random number generation and user selected timescales were implemented. Other development initiatives were investigated and either postponed or abandoned due to complexity of implementation.