Notes on the SimulTest Programs Used in the Language Selection Process

Tyche 3.0 Development Project

Terry Restoule
Contractor, LeverageTek IT Solutions

DRDC CORA CR 2012–092
May 2012

Defence R&D Canada
Centre for Operational Research and Analysis

Force Readiness Analysis Team
Force Readiness and Air Systems Section

National Defence Défense nationale

Canada

Notes on the SimulTest Programs Used in the Language Selection Process

Tyche 3.0 Development Project

Terry Restoule
Contractor, LeverageTek IT Solutions

Prepared By:
Terry Restoule
136 Lewis Street, Suite 1
Ottawa, ON
K2P 0S7
LeverageTek IT Solutions
Contractor's Document Number: [if used]
Contract Project Manager: Tony Tornberg, 613-238-5100x5
PWGSC Contract Number: W7714-3810
CSA: Cheryl Eisler, 613-947-9796

Defence R&D Canada – CORA

Contract Report
DRDC CORA CR 2012-092
May 2012

Principal Author

Original signed by Cheryl Eisler

Cheryl Eisler

Contract Scientific Authority

Approved by

Original signed by Denis Bergeron

Denis Bergeron

Section Head, Force Readiness and Air Systems

Approved for release by

Original signed by Paul Comeau

Paul Comeau

Chief Scientist

Defence R&D Canada – Centre for Operational Research and Analysis (CORA)

Notes on the SimulTest Programs Used in the Language Selection Process

Tyche 3.0 Simulation Engine Project

Prepared By:
Terry Restoule
LeverageTek IT Solutions
136 Lewis Street Suite 1, Ottawa, ON, K2P 0S7
Contract Number: W7714-3810
Contract Scientific Authority: Cheryl Eisler, Force Readiness Analysis Team, 613-947-9796
Published in May, 2012 as DRDC CR 2012-092

Version 1.0
Latest Revision
May 4, 2012

RECORD OF AMENDMENTS

Amendment No.	Entered By	Revisions	Date
1	Terry Restoule	Original Version	Oct 21, 2011
2	Cheryl Eisler	Minor Revisions	Nov 18, 2011
3	Terry Restoule	Minor Revisions, SimIO and Simul8 Data	Dec 1, 2011
4	Cheryl Eisler	Minor Revisions	Dec 14, 2011
5	Terry Restoule	Complete Table 1, Abstract and Summary	Dec 16, 2011
6	Cheryl Eisler	Minor revisions	Dec 21, 2011
7	Cheryl Eisler	Minor revisions	Jan 10, 2012
8	Cheryl Eisler	Minor revisions after Section Head review	Apr 2, 2012
9	Cheryl Eisler	Finalization of report	May 4, 2012

ABSTRACT

The objective of this work was to recommend a software platform to be used in the redevelopment of the Tyche Simulation Engine. Several different programming languages and simulation packages were evaluated for technical suitability and relative performance. A program template was developed and test programs were written for each platform. Testing was performed on a common computer. Visual C#.NET was shown to be the best fit overall.

RÉSUMÉ

Cet ouvrage vise à recommander une plateforme logicielle qui sera employée dans le redéveloppement du module Tyche SE (Simulation Engine). Plusieurs langages de programmation et logiciels de simulation ont été évalués quant à leur adaptabilité sur le plan technique et à leur rendement relatif. On a développé un modèle de programme puis écrit des programmes d'essais pour chacune de ces plateformes, qui ont été mis à l'essai sur un même ordinateur. Dans l'ensemble, Visual C#.NET s'avère être celui qui correspond le mieux aux besoins.

EXECUTIVE SUMMARY

The Tyche Simulation Engine and graphical user interface (GUI) programs were developed using Visual Basic 6 (SP6), which is no longer supported by Microsoft. To determine a successor to VB6, a number of commercially available and open source programming languages and software packages were selected as potential candidates. Candidates were limited to development environments with documented capability for discrete event simulation, parallel processing support, and use in a high-performance computing environment.

A subsequent 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 in each of the programming language platforms and simulation packages being examined. Strengths and weaknesses in each of these programs are outlined, along with descriptions of problems encountered. Timing information for each of the demonstration processes in each of the test programs were collected for multiple iterations. Table 1 provides average relative performance measures. Performance tests were conducted on a common computer.

All of the potential development languages generally outperformed Visual Basic 6 in their tests. Overall, Visual C#.NET ran the fastest. In the areas where other products were faster, Visual C#.NET was not far off the best times. Most of the other programming languages performed well (though performance was less impressive). The notable exception was Visual C++.NET. It suffered from having to conform to both the C++ standard and the .NET standard. This led to programming difficulties and slow performance.

Of the two simulation packages examined, SimIO performed well but the object-oriented three dimensional modeling environment would require a complete redesign of the Tyche Simulation Engine. Due to inadequacies in its Visual Logic language, the coding tests that were conducted with all of the other software platforms could not be performed for Simul8.

SOMMAIRE

Les programmes module Tyche SE (Simulation Engine) et d'interface graphique ont été développés à l'aide de Visual Basic 6 (SP6), pour lequel Microsoft n'offre désormais plus de soutien technique. Afin de déterminer quel sera son successeur, de nombreux produits commerciaux ainsi que des langages de programmation et des progiciels en source ouverte ont été sélectionnés comme candidats potentiels. Ceux-ci se limitaient à des environnements de développement possédant des capacités démontrées de simulation d'événements discrets, de prise en charge du traitement parallèle et pouvant être employés au sein d'un environnement informatique haute performance.

Par la suite, on a défini un ensemble de caractéristiques logicielles requises puis développé un modèle de programme en vue de faire la démonstration de ces caractéristiques. Il a d'abord été écrit à l'aide de VB6 afin qu'il soit possible d'effectuer le test de rendement.

Par la suite, des versions du programme modèle ont été développées avec chaque plateforme de langage de programmation et on a examiné chaque logiciel de simulation. Les forces et faiblesses de chacun sont décrites, accompagnées d'une description des problèmes survenus. Chaque programme d'essai a exécuté les processus présentés à plusieurs reprises, pour lesquelles les temps de traitement ont été consignés. Le tableau 1 indique la moyenne des mesures de rendement relatives de chacun de ces programmes. Ces tests de rendement ont été effectués sur un même ordinateur.

Au cours de ces essais, tous les langages de programmation potentiels affichaient un rendement supérieur à celui de Visual Basic 6 de façon générale. Dans l'ensemble, Visual C#.NET s'avère être le plus rapide. Dans les cas où d'autres produits était plus rapides, Visual C#.NET les suivaient de très près. La plupart des autres langages de programmation ont obtenu un bon rendement (qui était toutefois moins impressionnant), exception faite de Visual C++.NET. Il était désavantagé, car il devait respecter les normes C++ et .NET, ce qui a causé certains problèmes de programmation ainsi qu'un faible rendement.

Parmi les deux logiciels de simulation examinés, SimIO a obtenu un bon rendement, mais l'environnement de modélisation 3D orienté objet nécessiterait une refonte complète de Tyche SE. Il a été impossible d'exécuter avec Simul8 les essais de codage effectués avec toutes les autres plateformes logicielles, en raison des insuffisances présentes dans son langage Visual Logic.

TABLE OF CONTENTS

1. INTRODUCTION

The Tyche Simulation Engine and graphical user interface (GUI) programs were developed using Visual Basic 6 (SP6), which is no longer supported by Microsoft. It has been decided that future development should be conducted using fully-supported, contemporary software tools. A survey[1] was conducted of commercially available and open source programming languages and software packages for discrete event simulation, parallel processing support, and capability for use in a high-performance computing environment. The possible languages/packages that could support the functionality required in Tyche were narrowed down to the following:

- Microsoft Visual Studio with MS-MPI, including
  - Visual Basic.NET (VB.NET)
  - Visual C#.NET (VC#)
  - Visual C++.NET (VC++)
- SimIO
- AnyLogic
- Simul8
- SimEvents (Matlab)
- SimPy (Python)

In order to properly evaluate the various software development tools being considered for further Tyche development, each of the recommended tools were used to assess their suitability and performance.

A list of required programming features was given and a processing template was developed, from which test programs in each of the potential languages/packages were created.

2. FEATURES BEING DEMONSTRATED

The following features were requested:

- Set values to variables of various types (`Boolean`, `long`, `short`, `integer`, `string`, etc.);
- Mathematical calculations (`+ - / *`) ;
- Random number generation;
- Implement and Call class elements (.value), create and destroy;
- Adding, removing, searching and counting through a collection;
- Nested `if`/`else` and case statements;
- Loops (`for`, `while`);
- String manipulation;
- Creating, expanding and editing an array;
- Call a user-defined function;
- Force an error and capture it in an error handler;
- Utilize functionality similar to `Goto`, `exit`, `is()`, `cstr()`, etc.;
- Programs must work as console applications;

[1] Guo, R.J. and Brooks, J. (2012), Analysis of the Tyche Simulation Engine and Recommendations for Future Development, (DRDC CORA CR 2012-081), CAE Professional Services, Ottawa, ON.

- Program must zip the output file;
- Must allow for an input command line parameter (number of iterations);
- Programs must include timing code to the millisecond level for the entire program and for each function in an iteration; and,
- Functions must be set up such that processing times are measurable.

3. THE PROGRAM TEMPLATE

A template program was produced in Visual Basic 6. This was done so that there would be a benchmark program developed on the existing software platform, against which the other languages/packages could be compared.

The main program collects the run's start time, opens an output file and checks for a command line parameter. If present, this parameter value is used to set the number of times that the set of functional routines will run. If no parameter is passed, the program will perform 10 iterations.

The main processing loop is then entered and a label for the current iteration is written to the output file and the functional routines are run.

When the main processing loop has completed, the final timing information is written to the output file, the file is closed and the output file is compressed using a remote call to the WinZip command line processor.

The code used for WinZip processing was taken directly from the Tyche Simulation Engine v2.3.4. All other code in the template program was original.

The functional routines consist of the following:

- **LoadStr** - This routine produces a 15,000 character string of randomly generated lowercase letters. Timing information is collected before and after the string is generated and the string and its timing information is then written to the output file.
- **NumberCrunch** - This routine starts with a randomly generated number between 1 and 10. It then applies 20,000 randomly generated single digit numbers to it with randomly generated arithmetic operations for each. A string of the applied operations is maintained throughout the loop. Timing information is collected both before and after the operations are performed. When the calculations have completed, the string of operations and the timing information is written to the output file.
- **BookWordCount** - This and the following two routines perform the same function through different means. A large text file (the novel "Treasure Island") is read and value pairs of each unique word and the number of times the word appears are created. This routine uses a user-defined structure (containing a string and an integer value) and a dynamic array structure to store the word/count pairs. Each of the routines use the `CleanWord` function to convert each word found in the file to a lowercase expression and to eliminate all numerals and punctuation from the words. Each time a word is found in the text, it is cleaned and the array is searched for the same word. If the word is found, its count is updated and the array search is exited. If the word is not found, the array size is extended and the new word is added to the end of the array. Timing information is collected before and after the book is processed. When the book has been read, the contents of the array and the timing information is written to the output file.
- **ClassCollBookCount** - This routine processes the book file used by `BookWordCount` and builds a set of word count pairs in a similar way. Instead of using a user-defined structure, it uses a class object with a string Word property and an integer Count property. The word/count pairs are stored in a collection of those class objects. Each time a word is found in the text, it is cleaned and the collection is searched for the word. If the word is found, its count is updated and the search of the collection is exited. If the word is not found, a new member of the collection is created and the new

word is placed in that entry. Timing information is collected before and after the book is processed. When the book has been read, the contents of the collection and the timing information is written to the output file.

- **DictBookCount** - This routine processes the book file used by `BookWordCount` and builds a set of word count pairs in a similar way. Instead of using a user-defined structure and array, it uses a dictionary object. Dictionary objects store key/value pairs using a hash table. Here, the key is the word and the value is the word count. Each time a word is found in the text, it is cleaned and the dictionary is checked for the word. If the word is found, its count is updated. If the word is not found, it is added to the dictionary with a value of one placed in the location determined by the hashing function. Timing information is collected before and after the book is processed. When the book has been read, the contents of the dictionary object and the timing information is written to the output file.

Error and timing information is also written to the console throughout the processing. This is done through Windows API calls.

Timing values are collected using the `GetTickCount` Windows API call.

Error handling code is present in all major functions and subroutines.

4. THE TEST PROGRAMS

In order to demonstrate the required features listed in section 2, test programs in each of the potential development languages were written. The following sections deal with the process of creating each test program and what was experienced with each language/package. Required features that could not be demonstrated are identified (if present) and problems encountered with each language or package are discussed.

Final tests were run from a command window and not from within any program development environment to fairly compare performance (also demonstrating ability of environment to produce deployable packages, which are necessary for installation on a number of computers for large-scale batch processing). Though the number of major iterations in each program run could be controlled by an input parameter, the final tests performed 10 iterations (the default value).

Programming of the .NET languages was performed using Visual Studio 2010 Express (freely available from Microsoft). Other software versions include:

- Dev C++ 4.9.9.2 (open source)
- Python 2.7 with NumPy 1.6.1 and SimPy 2.1.0 (open source)
- SimIO Release 4 (evaluation)
- Simul8 2010 (evaluation)

AnyLogic was ruled out as a possible software package after consultation with DRDC CORA staff, as the development license requires drivers that are not acceptable for security reasons on the computer network. Restricting Tyche development and debugging to standalone computers was deemed a sufficient complication to remove AnyLogic from further consideration.

Matlab and Simulink R2010a (7.10.0.499) with Statistics and SimEvents Toolboxes licenses were only available on the DRDC CORA computer network and, as a result, the test program was written and run by the contract scientific authority. Similarly, the work done to modify the Python code to run in SimPy was

also conducted by the contract scientific authority. This work will be documented and compared with the test programs from the remainder of this document in a forthcoming DRDC Technical Memorandum.

4.1 *VISUAL BASIC .NET*

With VB.NET being the next generation of Visual Basic, this test program most closely resembled the template program. Much of the code was directly ported from the template and then adjusted accordingly. The VB6-style error handling was replaced by `Try/Catch` structures in each of the routines. File I/O was performed by `StreamReader` and `StreamWriter` objects provided by the `System.IO` namespace.

There were no problems until it came time to compress the run output. As in the template program, the code used to call WinZip remotely was ported directly from the Tyche Simulation Engine and used Windows API calls extensively to manage the remote process. At first, there were problems mapping the function declarations because the parameters and return values defined as "`Long`" in VB6 had to be defined as "`Integer`" in VB.NET because integers are 32-bit entities in VB.NET (versus 16-bit in VB6). With the declarations cleaned up, unhandled exceptions were generated through these API functions (attempted write to protected memory). Research determined that the problems were due to Windows API calls being unmanaged code. The Windows API approach to the remote process was replaced by the Process object provided by the `System.Diagnostic` namespace. The problems disappeared and the Windows API related code (a separate module in Tyche) was replaced by eight lines of code.

All Windows API calls were then replaced by functionally analogous objects internal to .NET. Timing information was retrieved using the `TickCount` method in the `System.Environment` namespace and the writes to the console were done using the Console object in the `System.Environment` namepace.

In Visual Basic 6, any program that wished to interact with a system console window (DOS command window) had to be linked to the system console object explicitly using the link.exe program that shipped with VB6. The new VB.NET program was created as a "Console Application" within .NET and therefore did not need to be linked separately.

VB6 code translates reasonably well to VB.NET and performance is quite good. Refer to section 6 for more details.

4.2 *VISUAL C#*

Surprisingly, converting from the VB6 template program to VC# was mostly straightforward. Much of the VB.NET code translated directly, with just the addition of a semicolon to the end of the lines. All of the .NET specific references worked the same as in VB.NET. Once the user adjusts to the look of the C-style language, the rest of the coding specifics came pretty naturally (braces, variable definitions, etc.). The error handling and WinZip processing (Process object as used in VB.NET) translated with minimal effort.

Class syntax is substantially simpler than in the VB world (though it took some doing to figure out what was needed). The only problem area encountered was in the handling of command line parameters. They are implemented differently in VC# than in either VB or C++.

The VC# program came together faster than any of the other test languages (faster even than VB.NET because there was no attempt made to use Windows API calls). Visual C# also demonstrated consistently the best performance. See section 6 for details.

4.3 *VISUAL C++ .NET*

After the ease of converting to Visual C#, using Visual C++.NET was very frustrating. The first clue that the level of effort would be significantly greater was upon discovery that managed variables (anything from a

.NET namespace) can only be defined locally. Only variables of simple types (`bool`, `int`, etc.) can have wider scope. This meant that many more parameters had to be passed to maintain accessibility.

The next discovery was that variables of type "`string`" and variables of type "`String`" are not the same, and are completely incompatible. This was just a symptom of a greater truth. The .NET specific world is considered "managed" and the C++ world (anything available through the standard C++ libraries) is considered "unmanaged". The two solitudes cannot coexist easily. The error message "No conversion exists in any context" got tiresome quickly. This led to a rather schizophrenic coding experience.

Before writing any portion of code, a decision had to be made as to whether to write the code as primarily C++ or whether .NET internals could be used. Anything that involved extensive string manipulation, for example, needed to be written in C++ (so that "`string`" variables could be used). The WinZip portion had to be written in .NET (with no C++ variables or calls) in order to use the Process object in the way that the other .NET languages had. The origins of the variable types even dictated how information was written to the output file or console. If the variables were C++ based, they had to be written using an `ofstream` or a `cout` statement; if .NET variables were used (e.g. "`String`"), then a `StreamWriter` object or the `System Console` object had to be used.

Constructs in the different worlds that performed similar functions did not always have similar performance. The first version of the LoadStr routine used a .NET `StringBuilder` object. The routine ran predictably and was not difficult to write, but the processing took 45 seconds per iteration (1,000 times slower than VC# and 3,000 times slower than VB6 in these test cases).

Things got really ugly when classes and collections came into play. If using the .NET Collection constructs, only managed classes could be used (i.e. only properties with simple types or .NET types). Since the routine involved significant string manipulation, it was written extensively with C++ strings. To rewrite the routine with `StringBuilders` could have been done, but the performance would have been horrendous.

With generally unimpressive results and continued programming difficulty, further work on the Visual C++.NET program was abandoned and other C++ implementations were investigated.

Following the work on the Dev C++ test program (see section 4.4 below), the Visual C++ test program was revisited and the collection and dictionary routines used in the Dev C++ program were copied and pasted into the VC++.NET program. Timing and error handling code were adjusted to accommodate the .NET specifics and the program was compiled. The performance was abysmal. The dictionary routine was 8 times slower than its nearest competitor and the class and collection routine took twice as long as in VB6.

4.4 *DEV C++*

Dev C++ is a free software development product built on the GNU C++ compiler. It comes with its own integrated development environment (IDE), a straightforward installation procedure and a configuration wizard. It can be used to create a wide variety of applications including Windows applications, console programs and dynamically linked libraries (DLLs).

The editing environment is similar to the .NET environment, though the symbolic debugger takes a bit of getting used to. One criticism of the IDE is that it is not well documented. When trying to debug the program, it would complain that the program had to be recompiled with the debugging information included. What it did not say is that along with the "enable debugging" switch in the project properties, an undocumented compiler switch had to be entered. This was just one of the documentation problems that had to be overcome.

The C++ implementation is to standard and works well. There were no real surprises when coding and development of the test program went smoothly (IDE documentation issues and general acclimatization aside).

Though error handling is included in the code, it would appear that the only way to throw an exception is to include an explicit "throw" statement in the code.

By comparison to Visual C++.NET, Dev C++ is a joy to use but, that being said, the performance was strictly middle of the road. See Section 6 for details.

4.5 *PYTHON*

Python is an open-source, interpreted programming language. While all files are simple text-based scripts, a free tool[2] for supporting Python in the Microsoft Visual Studio 2010 IDE was found to assist with programming in a similar environment to the other test cases. Unfortunately, help support (using F1 or View Help) was not included.

Coding in Python is very compact and once the user gets used to the lack of `end` statements, coding is very quick. Many common functions are already written, which also simplified coding, although finding the appropriate functions and the right module to import was sometimes problematic. String manipulation and file handling were also simple to accomplish. The intrinsic Dictionary object provided excellent performance. The "`try/except`" function was also found to be significantly faster than iterating through Lists. The built in `zip` function and the ability to call shell commands were simple and useful.

Python however, was not without its problems. Several aspects of the language differ greatly from the other languages evaluated. Variables that were defined to have global scope in the other languages had to be defined inside each function that used them. There was no case/switch statement. Class structures were not easily understandable, especially with respect to overloaded operators. Error handling was also difficult to understand, especially when trying to retrieve all of the information that Python normally extracts from the call stack.

Unlike commercial programming environments, Python is open source. This results in a lack of versioning continuity and agreement about the best version to use. While Python 3.2.2 is the most recent version, older versions (such as Python 2.7 used here) are still in common use – depending on the preference of the user. While documentation exists for all versions, newer code is not always available with sufficiently worked examples to be useful. In addition, it was difficult to determine the exact combination of add-on site packages that would work together correctly.

Specific to the test case, the Dictionary object performed very well, but did not maintain the order of entry that the other languages did. Instead, the OrderedDict object had to be used, which was not nearly as efficient.

The lack of integrated help (the web documentation[3] was not always straightforward or easy to understand) and unfamiliarity with the language negated much of the primary advantage of selecting an interpreted language – that of ease and speed of implementation. Once coding and data use was optimized, performance was only a slight improvement over VB6.

4.6 *SIMIO*

Unlike the software development platforms discussed to this point, SimIO is not a general purpose programming environment. It is used specifically to develop discrete event simulations in a visual programming interface. Built on an object-oriented concept, it comes with a set of standard modeling objects that the user can "click and drag" to create models. These objects can, however, be modified by the

[2] http://pytools.codeplex.com/

[3] http://docs.python.org/tutorial/index.html

user or used as building blocks to create other objects as needed. In fact, everything that can be manipulated is an object (including whole simulations).

Probably SimIO's greatest strength is in its graphics engine. Three dimensional modeling environments can be created. Individual entities (fixed and movable) can be given three dimensional graphic symbols. Symbols can even be downloaded from Google's 3D Warehouse. The user's perspective can also be adjusted during design time and during runs. The environment can be zoomed in and out or rotated. Specific entities can be selected and "cameras" can be chosen allowing the user to follow the entity through the model.

SimIO also allows the user to create system extensions in the form of DLLs. Any .NET language can be used in creating these libraries. For the purposes of this evaluation, a user extension was written using VC#. The extension caused an iteration of the test processes to be run every time an "Entity" entered a "Work Center" control. A second extension was written that zipped the output file. This was executed through SimIO using an "end of run" event.

Though the C# code migrated to the DLLs with no changes, getting the code to be executed by SimIO properly took a lot of trial and error because the User Extension documentation is practically useless.

The performance of the code in the DLLs was surprising. It ran slightly faster from inside SimIO than the C# code did when executed directly from a command window. One possible explanation for the observance of this phenomenon when the test case was run on the test platform is that SimIO automatically optimizes itself when running on multi-processor computers (more than one CPU). The embedded parallel execution in SimIO may be responsible for the observed performance improvement. More testing would be required to determine if this were the case, and the extent to which the parallelization could improve code performance.

Given SimIO's development interface, it would be all but impossible to create a simulation model without a graphically-based and animated user interface. The current Tyche simulation has no such interface of any kind to be migrated. For this reason, re-creating Tyche in SimIO would require a complete conceptual restructuring of the model and a code redesign from the ground up. Once a visual motif for the system had been created, the nature of that motif (and the animation inherent in it) would influence the entire system's design.

SimIO is available in several commercial versions. The main single user version is called Design Professional, which is available for $11,900.00 USD. The Team Version provides a design copy of Design Professional and creates simulations that can be run through the free Evaluation version (the developer uses the Design Pro license and the users install the Evaluation version on their computers). The Team version costs $16,900 USD. Yearly maintenance contracts are also available: Design Professional costs $2,390 USD/year while the Team version costs $3,390 USD/year.

4.7 *SIMUL8*

Like SimIO, Simul8 is specifically designed to create discrete event simulations and is not a general programming language. Also like SimIO, it has a click and drag visual programming interface. Unfortunately, that's where the similarities end.

Both packages provide a base set of simulation construction objects. With Simul8, these are the only objects that can be used. New objects cannot be built. "Properties" can be created for any instance of an object; however, these properties are actually only text labels. The graphical niceties that are available in SimIO are nowhere to be seen here. There is minimal 3D animation that consists of red balls cascading through each of the objects (that appear as boxes suspended above a hardwood floor).

All of that would be excusable if the modeling environment worked well, but it does not. The documentation had to be visited anytime the "convenient control key sequences" were attempted. The most unfortunate aspect of the creation process however was the way queuing was handled at each object. In order for the

"Work Station" objects to work at all, they had to be preceded by a "Storage Bin" object. These bin objects acted as the queue for the work station object. The problem is that visually, the bins looked and behaved like holding tanks. If someone is trying to graphically model anything other than fluids, this motif is bizarre. In SimIO, entities simply line up behind the work center they are waiting to interact with. If the Simul8 models let the red balls line up and wait, it would be much better than displaying liquid in a tank.

Two programming extension options are provided. First, extensions can be written in VBA (Visual Basic for Applications). Though offered, this interface is no longer supported. Most of the Visual Basic code from the test program could have been copied into an Excel workbook and executed as macros through Simul8 somehow, but would have provided performance no better (and likely worse) than Visual Basic 6. With the manufacturer not supporting it anymore, it did not seem worthwhile to explore or develop. The VBA interface has been replaced by a proprietary programming tool called Visual Logic.

To describe the Visual Logic programming interface as "cumbersome" is to be kind. Programming commands must be selected from a list to the left of the programming area. The commands are arranged into groups by function (basic commands, data, file, etc.). Within these groups, the commands are in no particular order. Once a command is selected, parameters are derived for the statement through a series of dialog boxes specific to each command. If the programmer needs to use a variable in a statement, the variable must have been defined (through another set of dialogs) in the global "Information Store" prior to creating the statement or all of the dialogs associated with the statement must be closed, the variable defined and the statement attempted again. With practice, a programmer would have to rewire their thinking so that these processes could become second nature.

Visual Logic code can be written in standalone chunks or it can be associated with objects in the simulation. The association is performed through dialogs handling the object's properties. The association must also be made before any coding is attempted.

Visual Logic does support SQL through ODBC connections, but does not support record definitions otherwise. It also does not support class objects or any data structures more sophisticated than dynamically sized scalar arrays.

With the absence of the necessary data constructs, as well as most of the common string manipulation functions, the word count procedures in the test program could not be implemented. The string loading procedure could not be readily constructed either. With time, some of these missing intrinsics could have been fabricated from Visual Logic but, with only a 14 day trial evaluation version, it was not possible.

Simul8 is available in several versions. Simul8 Standard is $1,495.00 USD. Simul8 Professional is $4,995.00 USD. The Simul8 web site is unclear as to what the differences are between the two packages but the trial evaluation version provided was identified as Simul8 Professional.

A network licensing arrangement is also offered. It is a concurrent user arrangement that requires a minimum purchase of 5 copies (Standard or Professional) x 1.2 of the cost. For a Professional network license, that would be ($4,995.00 x 5) x 1.2 = $29,970.00 USD.

Annual maintenance plans are also offered. For Simul8 Standard, maintenance costs $495.00 USD/license. Simul8 Professional maintenance costs $899.00 USD/license.

5. THE TEST PLATFORM

Final tests runs for all of the programs were performed on an Acer Aspire M3910 with a 3.2 GHz Intel Core i5-650 CPU (dual core with hyperthreading and turbo boost, 4 MB L3 cache, 3.20 GHz, DDR3 1333 MHz) with the Intel H57 Express Chipset, and 6 GB of RAM.

The test computer ran Windows 7 Home Premium (64 bit), Service Pack 1.

6. RESULTS

Table 1, below, summarizes the results of the final test runs of each of the programs. The numbers represent the average length of time the program spent performing a single iteration of the identified function (each test run consisting of 10 iterations except where noted). Refer back to section 3 for descriptions of each of the functions.

All of the potential development languages generally out performed VB6 in their tests. Each language had its own strengths and weaknesses. Overall, VC# ran the fastest. In the areas where other products were faster (LoadStr, Dictionary), VC# was not far off the best times.

An "Ordered Dictionary" process was added to both of the Python-based programs so that the dictionary output could be demonstrated as consistent with the output from the other programs. In Matlab, an alternative data structure, the cell array[4], was also used for performance comparison.

The following problems were encountered that could not be resolved in the time given:

- Due to inadequacies in the Visual Logic language, the coding tests that were conducted with all of the other software packages could not be performed for Simul8.
- Due to a glitch in the Visual C++ .NET program, the Total Run Time value is not available.
- When running the SimIO tests, the entities did not always stay in the work centre control long enough to completely append their results to the output file. The numbers in the SimIO column are taken from 10 recorded iterations made during several runs of the model.

[4] The cell array allows for storage of differing data types, such as strings and numbers, inside a single array.

Table 1: Average iteration run time results.

	Programming Language/Software Package Run Time (minutes:seconds)								
Function	**VB6**	**VB.NET**	**VC#**	**VC++**	**Dev C++**	**Python**	**SimIO**	**SimPy**	**Matlab**
Load String	0.016	0.045	0.053	0.042	0.017	0.014	0.061	0.014	0.007
Number Crunch	0.013	0.005	0.005	0.005	0.002	0.117	0.005	0.117	0.586
Array/List	2.649	1.250	0.627	5.839	1.272	1.151	0.487	1.139	1.380
Dictionary	0.183	0.059	0.056	1.670	0.197	0.032	0.052	0.032	1.418
Ordered Dictionary/ Cell Array	N/A	N/A	N/A	N/A	N/A	0.117	N/A	0.117	1.775
Class and Collection	22.252	2.136	1.111	44.856	3.717	21.534	0.685	21.316	1.157
Total Run Time	4:11.505	0:35.220	0:18.658	N/A	0:52.245	3:49.969	N/A	3:47.648	1:38.249

7. RANKING

Based on a combination of performance and technical appraisal, the software packages evaluated in this exercise (not including Matlab) are ranked as follows:

1. **Visual C#**

   This was the language that performed consistently best overall. It also lent itself well to migrating code from Visual Basic. It is reliable and works well with its development environment.

2. **VB .NET**

   VB .NET's performance numbers were not as impressive as C# but they are still generally respectable. It shares the same IDE as VC# and seems equally reliable. Since it is the next generation of Visual Basic, migrating from the old platform would be the most straightforward.

3. **Dev C++**

   This development environment works surprisingly well and the price certainly cannot be beat. Based on the open-source GNU C++ compiler, it is also rock solid for reliability. The implementation of C++ also seems to be very much to the accepted standard. Performance was quite good.

4. **Python 2.7**

   The Python language is very compact and surprisingly powerful (if a bit cryptic). With some perseverance, moderate performance can be obtained.

5. **SimIO**

   If judged purely on performance, power, and general fit and finish, this simulation package should probably be ranked higher. Apart from the cost, the fact that using SimIO would require a complete redesign of Tyche places it here on the list.

6. **Visual C++ .NET**

   Considering that this is a .NET language, it is disappointing that it has to be placed here. Whereas Microsoft seems to have built VC# to best use the .NET environment, they have made many

unfortunate design decisions here. Because it is C++, this language must support the standard. Because it is a .NET language, it must live within the .NET framework. As a result, it straddles the "managed" and "unmanaged" worlds the most painfully. Not only is it painful and frustrating to use, in the end, the performance is not good.

7. **Simul8**

   This package has nothing to recommend it. The visual aspects are crude and dated in appearance. The programming tools are all but unusable, and the data support is firmly rooted in Microsoft Office circa 1995.

Note that these rankings are purely based on the requirements identified in Section 2. An appraisal of the development environment's ability to support parallel processing, ease of use in a high-performance computing environment, and the effects of code debugging and maintenance will be included in an overall assessment to be detailed in a DRDC CORA Technical Memorandum[5].

[5] Eisler, C. (2012), Tyche 3.0 Development: Comparison of Programming Languages and Software Packages for an MCDES, (DRDC CORA TM DRAFT).

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Terry Restoule
136 Lewis Street, Suite 1
Ottawa, ON
K2P 0S7

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED
(NON-CONTROLLED GOODS)
DMC A
REVIEW:GCEC JUNE 2010

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

Notes on the SimulTest Programs Used in the Language Selection Process: Tyche 3.0 Development Project

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

Restoule, T.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
May 2012	22	0

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Contract Report

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – CORA
101 Colonel By Drive
Ottawa, Ontario K1A 0K2

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
	W7714-3810

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
	DRDC CORA CR 2012-092

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

The objective of this work was to recommend a software platform to be used in the redevelopment of the Tyche Simulation Engine. Several different programming languages and simulation packages were evaluated for technical suitability and relative performance. A program template was developed and test programs were written for each platform. Testing was performed on a common computer. Visual C#.NET was shown to be the best fit overall.

Cet ouvrage vise à recommander une plateforme logicielle qui sera employée dans le redéveloppement du module Tyche SE (Simulation Engine). Plusieurs langages de programmation et logiciels de simulation ont été évalués quant à leur adaptabilité sur le plan technique et à leur rendement relatif. On a développé un modèle de programme puis écrit des programmes d'essais pour chacune de ces plateformes, qui ont été mis à l'essai sur un même ordinateur. Dans l'ensemble, Visual C#.NET s'avère être celui qui correspond le mieux aux besoins.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Tyche; Programming Language; Discrete Event Simulation Software; Test Case;

Defence R&D Canada

Canada's leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

DEFENCE R&D DÉFENSE

www.drdc-rddc.gc.ca