Tyche 3.0 Development

Comparison of Development Environments for a Monte Carlo Discrete Event Simulation (MCDES)

Cheryl Eisler
Force Readiness Analysis Team

DRDC CORA TM 2012–231
September 2012

Defence R&D Canada
Centre for Operational Research and Analysis

Force Readiness Analysis Team
DRDC CORA

National Defence Défense nationale

Canada

Tyche 3.0 Development

Comparison of Development Environments for a Monte Carlo Discrete Event Simulation (MCDES)

Cheryl Eisler
Force Readiness Analysis Team

Defence R&D Canada – CORA

Technical Memorandum
DRDC CORA TM 2012-231
September 2012

Principal Author

Original signed by Cheryl Eisler

Cheryl Eisler

FRAT

Approved by

Original signed by Dr. R. E. Mitchell

Dr. R. E. Mitchell

Head Maritime and Air Systems OR

Approved for release by

Original signed by Paul Comeau

Paul Comeau

Chief Scientist

The information contained herein has been derived and determined through best practice and adherence to the highest levels of ethical, scientific and engineering investigative principles. The reported results, their interpretation, and any opinions expressed therein, remain those of the authors and do not represent, or otherwise reflect, any official opinion or position of DND or the Government of Canada.

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

Abstract

Tyche is a Monte Carlo Discrete Event Simulation (MCDES) for analysis of force structures. To date, Tyche has been implemented in Microsoft Visual Basic 6. Due to the fact that Microsoft no longer supports this version of Visual Basic, as well as the need to increase both simulation speed and capability, an Applied Research Project (ARP 11ic) was started to rewrite the Tyche tool in a new programming language and build upon its functionality. The first phase of the project was to identify and select a suitable programming language or software package that could support the functionality of the current program, as well as future development for parallel processing in a high-performance computing environment. After a survey of existing programs and open literature, six potential development environments were identified with the highest potential to meet the functional requirements necessary to support Tyche: Microsoft Visual Studio.NET, Simio, AnyLogic, Simul8, SimEvents (in Matlab), and SimPy (in Python). To determine the best development environment, an iterative test program was created to compare the speed and ease of implementation on the kinds of computations that Tyche routinely performs. Results indicated that Microsoft Visual C#.NET is the fastest for computation of the test case calculations, as well as the most flexible and easy to use development environment (given the history of Tyche and its developers). A variety of data types and structures were also compared for performance to make recommendations for future implementation.

Résumé

Tyche, destiné à l'analyse des structures de force, est une simulation d'événements discrets Monte-Carlo mise en œuvre à ce jour en Microsoft Visual Basic 6. Or, comme Microsoft n'offre plus aucun soutien technique pour cette version de Visual Basic et qu'il est maintenant nécessaire d'augmenter tant les capacités que la vitesse des simulations, nous avons lancé un projet de recherche appliquée (PRA 11ic) afin de réécrire l'outil Tyche dans un autre langage de programmation et, en même temps, d'y ajouter de nouvelles fonctions. Dans la première phase du projet, nous avons voulu choisir un langage de programmation ou une trousse logicielle se prêtant bien tant aux fonctions actuelles de Tyche qu'aux fonctions prévues (traitement en parallèle dans un environnement informatique à haut rendement). Après avoir étudié les programmes existants et la documentation librement accessible, nous avons cerné les six environnements de développement les plus susceptibles de répondre aux exigences fonctionnelles voulues pour prendre en charge Tyche : Microsoft Visual Studio.NET, Simio, AnyLogic, Simul8, SimEvents (dans Matlab), et SimPy (en Python). Nous avons ensuite créé un programme d'essai itératif afin de choisir l'environnement le plus propice en comparant la vitesse d'exécution et la facilité de chacun à mettre en œuvre le type de calcul le plus courant dans Tyche. Les résultats obtenus indiquent que Microsoft Visual C#.NET produit les exécutables les plus rapides pour les calculs d'essai et se révèle être l'environnement de développement le plus souple et le plus convivial, vu le passé de Tyche et de ses programmeurs. Nous avons aussi comparé le rendement de nombreux types et structures de données afin de formuler des recommandations pour la mise en œuvre.

This page intentionally left blank.

Executive summary

Tyche 3.0 Development: Comparison of Development Environments for a Monte Carlo Discrete Event Simulation (MCDES)

Cheryl Eisler; DRDC CORA TM 2012-231; Defence R&D Canada – CORA; September 2012.

Introduction: Tyche is a Monte Carlo Discrete Event Simulation (MCDES) for analysis of force structures. It generates a large number of equal time periods in which realistically random, concurrent, and geographically distributed scenarios are scheduled. Assets are assigned to these scenarios following a set of predefined rules in an attempt to reproduce the decisions of a real fleet scheduler. By repeating the process with different fleet compositions, the average performance of each candidate fleet can then be compared. To date, Tyche has been implemented in Microsoft Visual Basic 6. Due to the fact that Microsoft no longer supports this version of Visual Basic, as well as the need to increase both simulation speed and capability, an Applied Research Project (ARP 11ic) was started to rewrite the Tyche tool in a new programming language and build upon its functionality. The first phase of the project was to identify and select a suitable programming language or software package that could support the functionality of the current program, as well as future development for parallel processing in a high-performance computing environment. After a survey of existing programs and open literature, six potential development environments were identified with the highest potential to meet the functional requirements necessary to support Tyche: Microsoft Visual Studio (specifically composed of Visual Basic.NET, Visual C#.NET, and Visual C++.NET), Simio, AnyLogic, Simul8, SimEvents (in Matlab), and SimPy (in Python). To determine the best development environment, an iterative test program was created to compare the speed and ease of implementation on the kinds of computations that Tyche routinely performs.

Results: When the computational performance and qualitative impressions of each development environment are combined, the best choice for Tyche is VC#.NET. It is the fastest performing code on the test case developed herein and has the best suite of supporting features. Code conversion will take time, but would not seem to be as much of an issue as some of the other development environments. Code debugging and maintenance will be relatively straightforward. The VC#.NET integrated development environment (IDE) provides comprehensive debugging tools and help support. Since the 2010 VC#.NET Express version from Microsoft is free to download, there is no additional cost to the Tyche ARP to write code or run executable files. All of the .NET languages provide support for parallel processing using Microsoft Message Passing Interface (MS MPI); ensuring future programming needs for operation in a high performance computing environment will be met.

Significance: Converting to a modern development environment brings the advantages of manufacturer support, new IDE features, built-in parallel processing tools, and increased speed of operation. Until code conversion is completed and debugged, it is not known exactly how much the new programming language will affect the performance of the program. The VC#.NET test case showed a 96% reduction in run time compared to VB6. Extrapolating to a full simulation run

on a complex data set, individual simulation runs that previously took 3 hours to run could be reduced to less than 8 minutes. Although the same scaling is not likely, a significant improvement in performance is still expected.

Future plans: The second phase of the Tyche 3.0 development project is currently underway to convert the existing simulation engine from VB6 to VC#.NET. Once the code has been rewritten and debugged, a performance comparison with the current version (Tyche 2.3.4) will be conducted. Subsequently, more advanced functionality will be added to Tyche to broaden the scope of problems the model can handle. First, a user-defined timescale will be developed to model assets with shorter usage patterns. By allowing different timescales in the simulation window, greater fidelity can also be achieved for assets with shorter usage patterns, such as aircraft, unmanned vehicles, and personnel. To help mitigate risk in a future force structure, the Tyche scheduler can be improved to better mimic the human decision-making process. User-defined heuristics and assignment rules will better model the way a real scheduler assigns assets to different events, rather than treating all platform types equally; this allows for greater flexibility and realism for risk mitigation through more sophisticated mission planning schemes. File output formats will also be customized to tailor data specific to that required for analysis and visualization. In addition, the second phase includes plans to redesign the event handler to allow for concurrent operations (such as force generation tasks completed en route to theatre for force employment, common practice for naval crews), waypoints (e.g. resupply, transfer, or forward deployment of assets), and additional locations for asset events (such as allowing critical maintenance to occur at the nearest base vs. home base).

Sommaire

Tyche 3.0 Development: Comparison of Development Environments for a Monte Carlo Discrete Event Simulation (MCDES)

Cheryl Eisler ; DRDC CORA TM 2012-231 ; R & D pour la défense Canada – CARO; septembre 2012.

Introduction: Tyche, destiné à l'analyse des structures de force, est une simulation d'événements discrets Monte-Carlo qui génère un grand nombre de périodes temporelles égales où sont prévus des scénarios aléatoires et simultanés distribués géographiquement de façon réaliste. Les ressources sont attribuées à ces divers scénarios en fonction d'un ensemble de règles préétablies qui visent à reproduire les décisions réelles d'un planificateur de la flotte. Répéter ce processus avec diverses compositions de la flotte, on peut comparer le rendement moyen de chaque flotte ainsi évaluée. Tyche a été mis en œuvre à ce jour en Microsoft Visual Basic 6. Or, comme Microsoft n'offre plus aucun soutien technique pour cette version de Visual Basic et qu'il est maintenant nécessaire d'augmenter tant les capacités que la vitesse des simulations, nous avons lancé un projet de recherche appliquée (PRA 11ic) afin de réécrire l'outil Tyche dans un autre langage de programmation et, en même temps, d'y ajouter de nouvelles fonctions. Dans la première phase du projet, nous avons voulu choisir un langage de programmation ou une trousse logicielle se prêtant bien tant aux fonctions actuelles de Tyche qu'aux fonctions prévues (traitement en parallèle dans un environnement informatique à haut rendement). Après avoir étudié les programmes existants et la documentation librement accessible, nous avons cerné les six environnements de développement les plus susceptibles de répondre aux exigences fonctionnelles voulues pour prendre en charge Tyche : Microsoft Visual Studio.NET (composé plus précisément de Visual Basic.NET, Visual C#.NET, et Visual C++.NET), Simio, AnyLogic, Simul8, SimEvents (dans Matlab) et SimPy (en Python). Nous avons ensuite créé un programme d'essai itératif afin de choisir l'environnement le plus propice en comparant la vitesse d'exécution et la facilité à mettre en œuvre le type de calcul le plus courant dans Tyche.

Résultats : En tenant compte de la vitesse de calcul et des impressions qualitatives de chacun des environnements, le meilleur choix pour Tyche se révèle être Visual C#.NET. Cet environnement a créé le code le plus rapide dans notre étude de cas, et il propose le meilleur ensemble de fonctions d'appui. Porter le code prendra certes du temps, mais cela devrait être moins problématique qu'avec certains des autres environnements de développement mis à l'essai. Le débogage et la maintenance du code seront assez simples, car l'environnement de développement intégré de Visual C#.NET propose des outils de débogage complets et une aide en ligne exhaustive. En outre, comme Microsoft offre en téléchargement gratuit Visual C#.NET Express, version 2010, programmer et exécuter les programmes n'entraîne aucune dépense supplémentaire pour le PRA de Tyche. Tous les langages de la suite .NET prennent en charge le traitement parallèle à l'aide de l'interface Microsoft Message Passing Interface (MS MPI); cet environnement répondra donc aussi aux besoins prévus d'exécution dans un environnement informatique à haut rendement.

Importance : Passer à un environnement de développement moderne permet de tirer parti du soutien technique du fabricant, des nouvelles fonctions qu'apporte un environnement de développement intégré, de l'intégration des outils de traitement parallèle et d'une exécution plus rapide des logiciels résultants. Il nous est impossible de savoir exactement à quel degré le nouveau langage de programmation améliorera le rendement du programme avant de l'avoir porté et débogué intégralement. Cependant, les essais ont démontré avec Visual C#.NET une réduction du temps d'exécution de 96 % par rapport à Visual Basic 6. Extrapoler ces résultats à l'exécution complète d'une simulation sur un ensemble de données complexe laisse prévoir qu'une simulation qui prenait auparavant 3 heures pourrait être exécutée en moins de 8 minutes. Quoiqu'une telle accélération de Tyche soit peu probable, nous prévoyons tout de même une amélioration marquée du rendement.

Perspectives : La 2^{e} phase du projet de développement de Tyche 3.0 est en cours; elle vise à porter le moteur de simulation existant de Visual Basic 6 à Visual C#.NET. Après avoir converti et débogué le code, nous comparerons le rendement de cette version et celui de la version actuelle (Tyche 2.3.4). Cela fait, nous ajouterons à Tyche des fonctions plus avancées qui permettront d'élargir la portée des simulations que le modèle peut prendre en charge. Tout d'abord, nous ajouterons une échelle de temps paramétrable, ce qui permettra de modéliser des ressources dont l'utilisation est plus courte. Ajouter plusieurs échelles de temps à la simulation permettra de mieux modéliser les ressources dont l'utilisation est plus courte, comme les aéronefs, les véhicules et le personnel. Afin d'aider à atténuer le risque pour une structure de force utilisée à l'avenir, nous avons l'intention d'améliorer le planificateur de Tyche pour qu'il reproduise encore mieux le processus décisionnel humain. En permettant à l'utilisateur de paramétrer l'heuristique et les règles d'affectation, nous pourrons mieux modéliser comment les planificateurs de la flotte attribuent les ressources à divers événements plutôt que traiter indifféremment tous les types de plates-formes. Tyche sera ainsi plus souple et plus réaliste, et aidera à mieux atténuer le risque, car il prendra en charge des scénarios de planification de missions bien plus complexes. Le format des fichiers de sortie sera aussi modifié afin de ne comporter que les données nécessaires pour l'analyse et la visualisation. Outre ces plans, la 2^{e} phase vise aussi à repenser le gestionnaire d'événements pour permettre l'exécution simultanée de tâches différentes, comme des tâches de mise sur pied de la force en route vers le théâtre où elle sera déployée (pratique courante pour les forces navales), à ajouter des points de cheminement (ravitaillement, transfert ou déploiement des ressources) et d'autres emplacements où des événements peuvent toucher les ressources (par exemple, permettre d'effectuer la maintenance critique à la base la plus proche plutôt qu'à la base d'origine).

Table of contents

List of figures

List of tables

Acknowledgements

The author would like to thank Stephen Okazawa for his assistance with Python and ORIGAME, Ramzi Mirshak for his assistance with Python and SimPy, and Daniel Wojtaszek and Greg Hunter for SimEvents.

A significant portion of the programming work in Microsoft Visual Studio, Simio, and Simul8 was conducted by Terry Restoule (Contractor) and published under DRDC CORA CR 2012-092 [1].

This page intentionally left blank.

1 Introduction

1.1 Background

Tyche is a software tool that was first developed by Defence R&D Canada's Centre for Operational Research and Analysis (DRDC CORA) in 2004 to compare the performance of various naval fleet options. The goal was to provide a capability-based model that would schedule fleet assets (supply) to best attempt to meet a set of force generation/force employment missions (scenarios as demand), utilizing the rules a naval fleet scheduler would typically employ. The scheduling model was implemented as a discrete event simulation with stochastic input, simulating a large number of time periods in which realistically random, concurrent and geographically distributed missions are scheduled.

1.1.1 Version history

Tyche Version 1.0 [2] was used for the first naval Fleet Mix Study (FMS I) to identify the minimum number of assets in each proposed ship class that met a maximum acceptable political risk threshold [3]. Because of the length and difficulty of running simulations at the time, only 100 fleets were examined in the space of several months. Tyche was significantly overhauled to produce Version 2.0 [4], which expanded the scope of the model beyond simple naval ships to a joint, modular, capability-based environment. Subsequent upgrades [5-7] provided the ability to queue and manage a large number of simulations on multiple core processors, significantly reducing simulation management and run time. Tyche Version 2.2 was primarily used to conduct the second FMS in 2007-2008 [8], where over 1,000 fleets were examined in a three month time-frame. Tyche 2.3.4 is the current version.

1.1.2 Performance issues

To date, the Monte Carlo Discrete Event Simulation (MCDES) engine and graphical user interface for Tyche have been implemented in Microsoft Visual Basic 6 (VB6). Running a single fleet in a simulation is computationally intensive, requiring anywhere from 10 minutes to 24 hours to run (depending on the complexity of the model). Given a limited number of computers on which to perform simulations, there is a practical limit on the number of fleets that can be examined in a reasonable period of time for a research study. During performance testing on various hardware configurations [9], it was determined that performance was primarily dependent on the number of central processing unit (CPU) cores available, the CPU speed, and the use of advanced hardware technologies such as hyperthreading and turbo boost. The amount of computer random access memory (RAM) was not a factor with current technology (as a single simulation only used about 15 MB of RAM).

Due to the fact that Microsoft no longer supports VB6, and the need to increase both simulation speed and capability, an Applied Research Project (ARP 11ic) began in 2008 to rewrite the Tyche tool in a new programming language and build upon its functionality.

1.1.3 Software development plan

The first phase of the project is to test and select a suitable programming language or software package that could support the functionality of the current program, as well as future development for parallel processing in a high-performance computing environment. After a survey of existing programs and open literature, six potential development environments were identified [10]:

- **Group I:**
  - Microsoft Visual Studio.NET with Microsoft Message Passing Interface (MS-MPI), including
    - Visual Basic.NET (VB.NET)
    - Visual C#.NET (VC#.NET)
    - Visual C++.NET (VC++.NET)
  - Simio
- **Group II:**
  - AnyLogic
  - Simul8
  - SimEvents (Matlab)
  - SimPy (Python)

Each of these development environments meets the minimum functional requirements to support the features and computations currently performed in Tyche, as well as built-in tools for parallel programming, although Group I environments were recommended over Group II environments due to compatibility with Microsoft Visual Basic. To improve the performance of the code, the fastest-running environment should be selected; however, speed of computation must also be balanced with ease of code conversion and debugging. In a multi-year development project with multiple programmers, the ease of use of the development environment can significantly reduce time to completion. Also, given the complex heuristic scheduling algorithms used within Tyche, the ease of debugging the code is vital to ensuring correct output.

1.2 Aim

The purpose of this study is to implement a test case program in each of the development environments to compare their speed and implementation on the kinds of computations Tyche routinely performs [1]. The environment with the best overall characteristics will be selected for the development of Tyche Version 3.0.

1.3 Scope

This study is not intended to recommend a particular development environment for general purpose simulation tools, nor is it to optimize the implementation of every test case within a given development environment; instead, the goal is to identify where the largest and easiest speed

improvements can be obtained. It is recognized that faster data structures, methods, or procedures may exist.

1.4 Outline

The design of the test case for each of the various development environments described in Section 1.1 will be explained in detail in Section 2. The hardware and software systems on which the test cases were runs are described in Section 4. Results are presented in Section 5, with performance and implementation discussions in Section 5.2. Section 6 concludes with the selection of the recommended development environment for Tyche 3.0.

2 Test case design

2.1 Requirements

The purpose of the test case is to mimic computations that Tyche routinely performs, but can be implemented in a small program (less than 500 lines of code, approximately), measured by internal timing functions (at millisecond resolution), and repeated to obtain average function speeds. Specifically, the test program must demonstrate the following:

- Assignment of values to variables of various inherent data types (Boolean, long, short, integer, string, etc.);
- Mathematical calculations (`+ - / *`) ;
- Random number generation;
- Implementation and call of class elements (`.value`), including initialization and finalization (garbage collection);
- 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;
- Calling of a user-defined function;
- Forcing of an error and capturing it in an error handler;
- Utilization of functionality similar to `Goto`, `exit`, `is()`, `cstr()`, etc.;
- Implementation as a console application;
- Zipping of the output file;
- Input of a command line parameter (number of iterations); and,
- Timing code to the millisecond level for the entire program and for each function in a given iteration.

Note that all requirements are given with reference to VB6 functions currently used in Tyche 2.3.4. The data structures and built-in functions may not necessarily be reflected in the other development environments, and the closest alternatives found may be implemented. Where differences exist, they will be noted in the next subsection.

No parallel programming constructs were tested, as this will be implemented in Tyche at a later date.

2.2 Implementation

The purpose of writing a specific test case in each of the development environments, rather than relying on case studies or recommendations from literature, is threefold. First, and of greatest importance, is to represent the kinds of computations and procedures currently available in Tyche. While the code is to be completely rewritten, it must first replicate the functionality and output of the current program. There are fundamental elements to the program that will not change, and the goal is to represent those in the test case. Secondly, it is important that all development environments of interest be tested on the same hardware and operating system software. This ensures a fair performance comparison. The third purpose is to subjectively evaluate the ease of use of each development environment in terms of conversion of existing code, creation of new code, and debugging features. As Tyche is a complex program undergoing multi-year development by several programmers (including defence scientists), the ease of use of the development environment cannot be entirely sacrificed for speed improvements.

The details of the test case procedures are documented in reference [1]. Here the focus will be on why specific procedures and data structures were selected, along with any issues that arose with implementation. Advantages and disadvantages of each development environment will be discussed along with the results in Section 5.2.

2.3 Test case template

The basic template for the test case was developed according to Figure 1. The test case starts by initializing a main function and a set of global variables to store data about the simulation time, error handlers, and the output file name. Error handling begins and the simulation timer is started (noting that the timer cannot capture the initial start-up of any program it is implemented in). The main function accepts an optional input parameter from the command line, so that the user may specify the number of iterations that the simulation will run. If the user does not enter an input parameter, 10 iterations are performed (default value used in testing here). Next, the output text file is opened for writing. The simulation then loops through the specified number of iterations, performing separate functions for string manipulation, arithmetic operations, file input/output (I/O), data storage, and data manipulation each time. Once complete, the total simulation time is recorded, the output file is closed and compressed, and the function terminates.

The overall structure of the test case template resembles the way the Tyche Simulation Engine operates, as shown in Figure 2 [11]. While the steps performed inside the various functions in the test case are not in the form of a discrete event simulation, they do mimic the kinds of individual functions called within Tyche.

At the start, global variables are used to store information about simulation timing, iterations, the output file, and error handling. With the exception of performance timing, which is specific to the test case, Tyche requires similar data accessible to all internal functions. While Tyche also stores additional data for global use, the various data structures tested within the “book word count” function are more representative of this.

```
Start function
        Initialize global variables
        Start error handling
        Start simulation timer
        Get input parameter
        Open output file
        Loop
                Load string function
                Number crunch function
                Book word count function (in several forms, using different data
                structures)
                Each of the above functions has the following form:
                        Initialize global variables
                        Start error handling
                        Start function timer
                        Do steps
                        End timer
                        Append to output file
                        End error handling
        End simulation timer
        Append to output file
        Close output file
        Zip output file
        End error handling
End function
```

Figure 1: Program template in pseudo code.

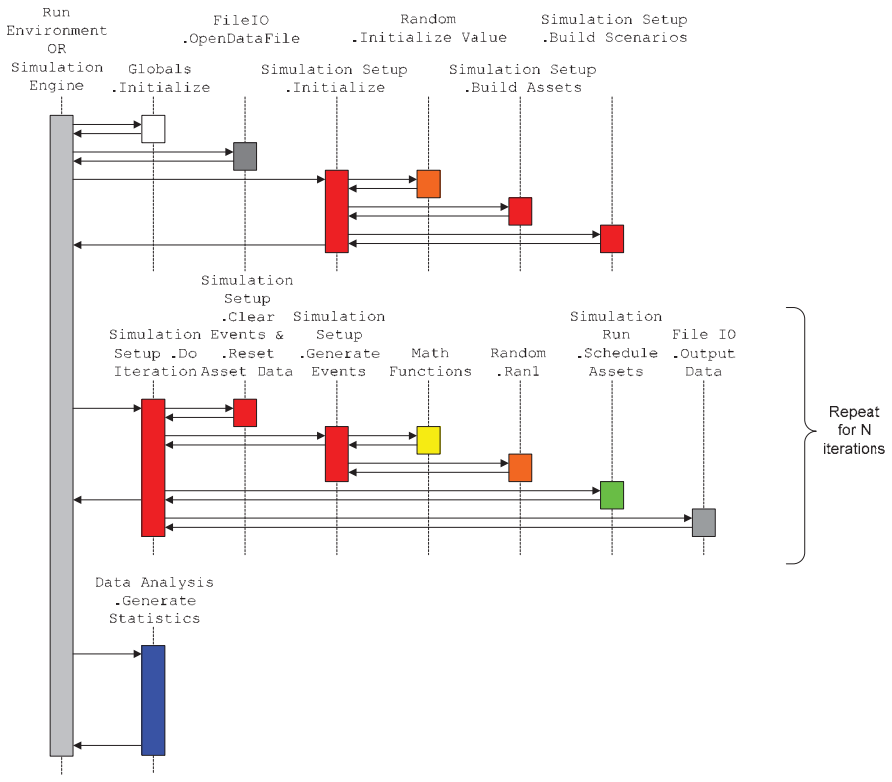

Figure 2: Tyche simulation engine functional diagram.

The test case then must accept input from the command line, for two reasons. The first is so that the Tyche Simulation Engine can be called from the command line with enough information to run a simulation, maintaining current functionality. The second is so that, in the future, the Simulation Engine can interact with job submission software in a high-performance computing environment.

Next, a standard ASCII text (.txt) file is created or opened for writing the individual function output and simulation performance time. Currently, the output from Tyche utilizes the same text file format, where the output from each simulation iteration and final statistics are written.

The test case then loops through the number of iterations provided either from the command line or based on an internal default value. Each iteration is comprised of at least three functions: "Load String" (for string manipulation), "Number Crunch" (for arithmetic operations), and "Book Word Count" (for file and data manipulation, implemented with one or more different data structures). Each internal function has been designed on the same general structure as the main test case function, with differences only in the steps performed. The following subsections detail each of the three function types.

When the simulation iterations are complete, the total run time is recorded. The output file is closed and then compressed using WinZip (or a similar program). The text file is deleted if the zip file is successfully created, error handling is wrapped up, and the function terminates.

Outside the main function, there are also separate functions defined for starting and stopping the run time clock, recording the run time, classes and structures to support the various data storage and manipulation functions, zipping the output, and, depending on the language, additional functions for file I/O and string parsing.

2.3.1 String manipulation

The first test function, "Load String", is quite simple. It generates a string of 15,000 random lowercase alphabetic ASCII characters. This requires the use of a *for* loop, a random number generator and string concatenation features. Both the randomly generated string and the function run time are written to the output file.

2.3.2 Arithmetic operations

The second test function, "Number Crunch", combines random number generation with arithmetic operations. A random starting integer between 1 and 10 is initially selected. 20,000 random arithmetic operations [+, -, *, /] are performed on the starting number sequentially using additional random integers [1,10]. The numeric operations are also converted into a string for recording in the output file, as well as the final number computed and function run time. This illustrates the additional features of a switch or case statement, a while loop, and data conversion from numeric to string formats (where applicable – not all languages support these functions or even require them explicitly). At the end of the "Number Crunch" function, an error is explicitly forced to demonstrate the ability of the program to capture and handle errors.

2.3.3 File and data manipulation

The third function, “Book Word Count”, can appear in three or four different variations in the test case, depending on the data structures available in a given development environment. The purpose of the function is to read in words from a book, stored as a simple ASCII text file, store the words as strings and count the number of times each word appears in the file. How the data is stored and the time it takes to read/write is of critical importance for the timing of this function, as Tyche stores and accesses a great deal of data with internal linkages. The word and count pairs are written to the main output file, as well as the function run time. This function demonstrates file input and output handling, if/else statements, and data storage and retrieval.

Specifically for the data manipulation, it is necessary to store the words from the book as a string and the associated count of the word appearance as an integer. The two pieces of information must be linked together, preferably in the same data structure. For example, two separate arrays could be generated, one to store the words and another to store the counts, with the array index matching between associated data. This is not an ideal data structure, as there is no explicit linkage between the information, making it difficult to debug the code and relying on knowledge resident in the programmer/user (rather than naturally embedded within the program).

Furthermore, the ability to retain the data based on the order in which it was entered (i.e., word appearance in the book) is of interest when comparing different data structures and development environments. This order of entry is utilized within Tyche, and not all data structures can necessarily support it. Different development environments can also support different kinds of data structures. Where possible, data structures with similar functionality were compared.

The data structures used in the test case include:

- type definitions/structures and arrays;
- dictionaries/maps;
- ordered dictionaries;
- cell arrays;
- classes and collections/lists; and,
- classes and vectors/dynamic arrays.

A type definition or a structure (`struct`) is a user-defined data type that allows the user to collect a number of different variables of various types (e.g., `integer`, `double`, `string`, etc.) that are related together. In the test case, two variables are defined: a “word” that contains a string and a “count” that contains an integer value of the word’s appearance in the text. However, this does not provide a way to conveniently store the entire set of words and their counts. This is where an array is required. An array contains multiple values of the same type. The test case implements an array of types or structures.

A dictionary or map is an inherent data structure that specifically combines the features described above. It contains multiple entries of keys and their associated values. The unique key for each element permits fast lookup, and dictionaries/maps often have additional predefined functions (such as adding or removing elements, counting the number of elements, etc.).

During the evaluation of the test case, it was found that some development environments handle dictionaries/maps in different ways. Most maintain the order of element entry; for example, a three element dictionary with the following key/value pairs entered respectively: house;3, car;5, box;1. Because dictionaries/maps are built on hash tables pointing to random memory elements, a few environments do not control how the data is retrieved. The data may come back randomly (box;1, house;3, car;5) or alphabetically (box;1, car;5, house;3). As a result, an ordered dictionary data structure was sought (Python 2.7 provides this type of inherent data structure).

An alternative to the ordered dictionary is the cell array (available only in Matlab). A cell array is like a normal array, except that the elements can store data of different types. As a result, a 2 by *n* cell array can be created, where the first dimension holds the "word" as a string, and the second dimension holds the "count" as an integer.

Finally, the most flexible data structure for collection of associated information is a class object. Classes are defined by the user and can hold variables of different data types, including other classes.[1] Classes are fundamental to object-oriented programming, and are used extensively throughout the current version of Tyche. For the test case, a class is created in the same fashion as the type definition/structure above. To provide the ability to store the set of words and their counts, collections/lists or vectors/dynamic arrays are used.

Collections or lists are inherent objects that allow the user to store and retrieve data on a group of other objects. Depending on the development environment and choice of collection/list type, they may be able to store heterogeneous or homogeneous data. For the purposes of the test case, only homogeneous collections/lists were used. They have a number of built in properties, similar to dictionaries, which allow for adding, removing, sorting, and counting of the objects they hold. They also support indexing, allowing elements to be accessed according to their position inside the collection/list, and maintain order of entry. Especially important is they grow and shrink dynamically as objects are entered or removed, requiring no memory pre-allocation.

Some development environments do not support collections or lists. In those instances, vectors or dynamic arrays are used. Vectors/dynamic arrays are similar to basic arrays, except that they can be expanded or shrunk after they have been defined (some environments require arrays to be fixed in length once defined, or only allow changes in one dimension). Section 3 will indicate which data types are specifically available in each environment.

2.4 Run procedure

Where possible, each test case is compiled into an executable file and called from the command prompt to fairly compare performance independent of the development environments. This also demonstrates the ability of the environment to produce deployable packages, which are necessary for installation on a number of computers for large-scale batch processing. In the cases where compiled executables cannot be generated, the development environments generally offer the ability to execute files with free "run-time" versions of the software for large-scale deployability; however, this was not specifically tested. Options for compilation are discussed in Section 3.

[1] One difference between classes and structures is that classes are passed by reference and stored on the heap while structures are passed by value and stored on the stack.

3 Notation on test case implementations

3.1 Development environments

The implementation of the test case in each development environment will be discussed. The development environments are broken down into two categories: programming languages and simulation software packages. The two categories provide distinctly different interfaces and approaches to simulation modelling. In addition, the development environments are listed in order from most similar to the current environment of VB6 to least similar, with the differences detailed in the subsequent subsections. This section is intended for computer programmers; those readers who are not interested in language details are recommended to proceed to Section 4 on hardware and software configuration. Note that the versions and sources of the software used are specified in Section 4.2.

3.2 Programming languages

High-level programming languages, such as those provided in the Microsoft Visual Studio package, are designed to make the process of developing a computer program easier. They feature:

- Abstraction from low-level machine code
- English-like commands
- Significant flexibility and control over data structures, logic, and flow

They are often part of an integrated development environment (IDE) that provides the user with a variety of features, such as a graphical user interface, debugging tools, context sensitive help, predictive typing, and a host of others, depending on the manufacturer. Programming languages are meant to be for broad-spectrum use and, as a result, often require some level of effort to learn the syntax (although this varies widely).

Once code is written, it is executed in one of three ways: compilation, interpretation, or translation. Code that is compiled forms a standalone executable (.exe) file to run. Interpreted code is read at run time, utilizing the development environment to execute the instructions – often with associated speed and memory overhead. Translated code is often used in languages that are written to be very high-level, with significant abstraction and ease of programming. The code is translated into another, more common programming language before being converted into an executable.

3.2.1 VB6

VB6 is a compiled language that was part of the Microsoft Visual Studio package, but is no longer supported. As a result, issues with ActiveX controls, Windows updates and maintainability arise. Since the Tyche Simulation Engine is currently written in VB6, the first test case was written in the same to act purely as a benchmark.

Most of the VB6 test case was straightforward, with a few specific exceptions. It was necessary to write functions to be able to handle command prompt input and output, remote calls to the WinZip command line processor (taken directly from the Tyche simulation engine version 2.3.4), checking if a file exists, and string "cleaning" (i.e., removal of undesired characters and fixing formatting).

Within the word count test functions, as described in Section 2.3, the following data types were used in VB6:

- Type definition and array (dynamic);
- Class and array (dynamic);
- Class and collection; and,
- Dictionary.

Note here that a combination of type definition and collection cannot be used, as type definitions are passed by value and collections are passed by reference. This results in the inability to add a type definition to a collection (attempting the `.Add` method causes the compile error "only public user defined types defined in public object modules can be used as parameters or return types for public procedures of class modules or as fields of public user defined types"), and eliminating the use of a `for each` loop (they cause the compile error "`for each` control variable must be variant or object").

3.2.2 VB.NET

VB.NET replaced VB6 in Microsoft Visual Studio, with a number of significant changes in the programming language. While an automated code conversion feature does exist, it cannot handle some of the data structures used in VB6, thus requiring the user to clean up the code manually. Most times, it is easier to simply rewrite the code in VB.NET from the start.

Much of the syntax is similar to VB6, so coding of the test case was easy. An important feature of VB.NET is the ability to import the predefined *System.IO* class (from the .NET namespace) for handling of command prompts and file checking. It was still necessary to create a short function for the remote call to WinZip and string cleaning, but both were simplified from the VB6 version.

Within the word count test functions, as described in Section 2.3, the following data types were used in VB.NET:

- Structure and array (dynamic);
- Class and array (dynamic);
- Class and collection; and,
- Dictionary.

Also of interest in the .NET environment is the ability to use `for each` loops on arrays, provided that the programmer does not attempt to write to the `for each` loop variable. If alteration of the variable is required, a typical *for* loop and counter must still be used.

3.2.3 VC#.NET

Syntactically, VC#.NET is a mix of VB and C++, with the .NET namespaces available. After the VB.NET code was written, the VC#.NET code came together very quickly. Just like the VB.NET version, the predefined *System.IO* class was again used for handling of command prompts and file checking. It is important to note that the command line arguments had to be passed in as inputs to the main test function, rather than as information obtainable from a system class object. It was also necessary to create a short function for the remote call to WinZip and string cleaning, but both were simplified from the VB6 version.

Within the word count test functions, as described in Section 2.3, the following data types were used:

- Structure and array (dynamic);
- Structure and list (from .NET *Systems.Collections.Generic*);
- Class and array (dynamic);
- Class and list (from .NET *Systems.Collections.Generic*); and,
- Dictionary.

Unlike VB.NET, a list of structures can also be used; however, there is one issue with implementation. When looping through a list of structures, the data within the structures can be read but cannot be modified directly; therefore, a temporary variable is necessary to write the information into and then assign the temporary variable to the structure.

An additional limitation of VC#.NET is that it does not have a built-in function for resizing of multidimensional arrays. Like the previous issue, the data must be copied to a new array of the desired size and the references reassigned. Alternatively, jagged arrays (an array with elements containing further arrays) can be used.

As a further point of comparison with VB, VC#.NET is a strongly typed language where the programmer is required to explicitly declare all variables and perform most conversions. Very little can be done implicitly, entailing slightly more effort on the part of the programmer.

3.2.4 VC++.NET

Unlike the ease of conversion of the test case to VB.NET or VC#.NET, writing the code in VC++.NET was significantly more difficult. The implementation of C++ in the .NET framework brings into conflict two different object models and their interaction at the machine level. “Managed” code is native to the .NET framework (such as .NET classes, VB.NET, or VC#.NET) and manages the execution within the Common Language Runtime (CLR). “Unmanaged” code is not native to the .NET framework and runs outside of the CLR – which is the case for most of the data structures and standard libraries within the C++ language. Mixing the use of the two comes at the cost of performance overhead [12] and compatibility issues.

Issues included [1]:

- Variables of managed data types (from the .NET namespace) could only be defined locally within a function or class. Only variables of simple types (e.g., `bool`, `int`, etc.) could have wider scope. This meant that many more parameters had to be passed into functions to maintain accessibility of data.
- Variables of type `string` (C++) and variables of type `String` (.NET) are not the same, and are incompatible. The origins of the variable types also dictated how information was written to the output file and system console. If the variables were C++ based, they had to be written using an `ofstream` or a `cout` statement; if .NET variables were used, then a `StreamWriter` object or the `SystemConsole` 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 string manipulation function 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 the `string` methods from C++ that later replaced it).
- If using the .NET *System.Collections* namespace, only managed classes could be included in the collection (i.e. only variables with simple data types or .NET types).

After initial programming difficulty, work on the Visual C++.NET program was suspended and other C++ implementations were investigated. Following the work on the Dev C++ test program (see Section 3.2.5 next), the Visual C++ test program was revisited and all of the word count functions used in the Dev C++ program were copied and pasted into the VC++.NET program. 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. Timing and error handling code were adjusted to accommodate the .NET specifics and the program was compiled.

Within the word count test functions, as described in Section 2.3, the following data types were used:

- Structure and array (dynamic);
- Structure and vector;
- Class and array (dynamic);
- Class and vector; and,
- Map.

It was noted that the map returned results alphabetically, not according to the order of entry. C++ does not contain a built-in data type that does this, and additional coding or use of an external class library[2] would have been required to retrieve the order of entry.

[2] Such as http://www.boost.org/doc/libs/1_39_0/libs/multi_index/doc/index.html.

3.2.5 Dev C++

Dev C++ is a free software development product built on the GNU C++ compiler. It comes with a moderately well-developed 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 code editing environment is similar to the .NET environment, although the symbolic debugger requires some time to adjust to. One criticism of the IDE is that it is not well documented. Otherwise, the C++ implementation is up to standard and works well. Coding and development of the test program went smoothly.

Within the word count test functions, as described in Section 2.3, the following data types were used:

- Structure and array (dynamic);
- Structure and vector;
- Class and array (dynamic);
- Class and vector; and,
- Map.

It was noted that the map returned results alphabetically, not according to the order of entry. C++ does not contain a built-in data type that does this, and additional coding or use of an external class library[3] would have been required to retrieve the order of entry.

3.2.6 Python

Two of the recommended software packages, SimPy and SimEvents, are actually add-ons to other programming languages: Python and Matlab (respectively). To implement the test case in either of these add-ons requires rudimentary knowledge in the basic programming language. As a further basis for comparison, it was decided to implement the test case in both Python and Matlab to determine if additional overhead or complexity existed between the basic language and the add-on package. The first case to be examined was Python.

Unlike the previously described programming languages, Python is an interpreted language. This means that the code is not compiled into an executable[4]; instead, the code is only converted to machine language at run-time. This is known to cause performance degradation, but the benefits are often seen to outweigh this cost. In the case of the Tyche software, it is necessary to determine the magnitude of the computational cost to be able to balance it against the advantages of: compact, easy to read code; decreased programming time; accessibility of code to programmers and users; and a significant body of open source code libraries.

[3] Ibid.
[4] To deploy such a program would require installation of the Python complier on any computer that would run the program. Since Python is open source (free), there are no licensing issues.

The standard Python libraries included functions that greatly simplified string cleaning, command line input/output, file handling, file zipping, and calls to external programs (like WinZip). Reduced syntactic constructs helped to balance out the initial learning curve requirements. Python also handles a significant number of situations that would throw errors in many other programming languages. As a result, the only way to generate an error (with functional code) is to raise an error.

Within the word count test functions, as described in Section 2.3, the following data types were used:

- Arrays (dynamic), noting that two separate arrays were used (as Python does not have a structure-type object, `tuples` are read-only once created, and the only other options are dictionaries or classes);
- Class and array;
- Dictionary; and,
- Ordered dictionary (`OrderedDict`, available in Python version 2.7).

When attempting to check if a word exits in the data structures used above, Python ran significantly faster when using a `try/except` statement on value assignment, rather than a *for* loop (`for each` loop functionality is accommodated in the `for` loop) with conditional statements. Of note, one disadvantage of Python is that it does not support a case or switch statement (only `if/else if` statements). Another is that there are no predefined list/collection objects, or structures/type definitions; hence, the reduced set of data type combinations above. In addition, the dictionary object did not maintain the order of entry that the other languages did (retained in the code for performance comparison). Instead, the ordered dictionary object had to be used.

Further aspects of the Python language also 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. Error handling was also harder to implement, especially when trying to retrieve all of the information that Python normally extracts from the call stack.

3.2.7 Matlab

Like Python, Matlab is also an interpreted language – with an emphasis on simple scripting files (m-files). The primary difference is that Matlab is focussed on the ability to perform complex mathematical computations, data analysis, and visualization. While not of significant use for the Tyche simulation engine itself, the kinds of analyses performed on the Tyche output would be aided by having the data available in a compatible format to the analysis environment.

The advantage Matlab has over Python is that m-files can also be compiled to an executable after automated conversion to C code (provided that one's licenses include the necessary Matlab Compiler toolbox and a compatible standalone C compiler to do so). The executable still requires a full Matlab license to run unless the MATLAB Compiler Runtime (MCR) is included in the configuration setup to create a standalone .exe file. Since DRDC CORA has the requisite licenses, two separate versions were tested: the interpreted m-files and a standalone executable.

When the test case was first written in Matlab, it followed much of the form of the standard programming languages. Loops and case statements were employed to add and update data elements individually and text files were read line by line. However, Matlab's strengths lie in the area of matrix math – whereas loops are known to be notoriously slow. In an attempt to better illustrate the computational power of Matlab, the test case was written a second time to utilize matrices (multi-dimensional arrays) and matrix functions whenever possible. While the code was not fully optimized to maximize the performance of the m-file, it does illustrate how alignment of the data and algorithms with the paradigm of the development environment can aid performance.

The standard Matlab libraries included functions that greatly simplified string cleaning, command line input/output, file handling, file zipping, and calls to external programs (like WinZip). Reduced syntactic constructs helped to balance out the initial learning curve requirements.

Within the word count test functions, as described in Section 2.3, the following data types were used:

- Structure and array (dynamic);
- Class and array (dynamic);
- Map; and,
- Cell array.

Like Python, there are no predefined list/collection objects; hence, the reduced set of data structure combinations above. The map in Matlab returned data in alphabetical order, rather than by order of entry, so the added combination examined used a cell array. A cell array is an array that can hold multiple types of data structures (such as a combination of numerics, strings, or objects). In the test case, the cell array is used like a map (dictionary), with one dimension containing the word string and the second dimension the word count integer.

3.3 Simulation software packages

The second category of development environments that were examined was software packages designed specifically for DES. These packages provide another level of abstraction from standard programming languages, often with GUI's for two- or three-dimensional visual programming. The extra level of abstraction from the code is the primary strength of these packages, making it easier for the user to envision what they are creating and faster to implement, as the user does not need to memorize syntax or perform compiler-level debugging. It can also create some difficulty if the process being developed does not strictly align to the objects within the DES model. This is a known issue, and the software packages often include an interface to a separate programming language to give the user more flexibility.

These packages are similar to interpreted programming languages in that they typically run models from inside their own environments, but often have free run-time versions that can be distributed on standalone computers.

3.3.1 SimPy

SimPy is simply an add-on site package for Python that has predefined classes and functions designed for discrete event simulation. Because the test case is not designed as a true DES, it was necessary to adapt it to fit within the DES construct. Essentially, each iteration through the test functions was defined within a process class. The iterations were sent to the default server resource (first in, first out), with a capacity of one process object at a time, for processing. The simulation was initiated with a run time of 100, but completed when the server was empty. The remainder of the main function, and each of the test functions, was exactly the same.

Many more classes and functions exist within SimPy for DES, but were not tested here.

3.3.2 Python and ORIGAME

While the code was being developed for Python 2.7, Stephen Okazawa provided significant programming assistance – given his familiarity using Python 2.6 for the prototype software known as ORIGAME under the Right Person, Right Qualifications, Right Place, Right Time Human Resources Technology Demonstration Project (R4 HR TDP). The R4 HR simulation software environment is planned to allow analysts to model and analyze different types of military resource management and business processes, at any desired/required scale and level of complexity [13]. Although the focus is on personnel for the R4 HR TDP, there is nothing restricting its use for the larger resource management picture, including equipment and materiel. As a result, existing models like Tyche and MARS could potentially be migrated to the R4 HR simulation software environment. The advantages of such a move include the flexibility that will allow models built within the environment to be easily modified and enhanced, scalability to allow creation of models of indefinite size, and integration to permit different models to be combined and interact with one another [13]. This will increase the fidelity of possible analysis, and allow models to be easily shared with colleagues – thus building new system models using different combinations of sub-models. The R4 HR simulation software environment holds a lot of potential for future model development within CORA and DGMPRA, and the opportunity to test Tyche-like computations in the prototype environment will provide insight into future use.

The prototype is at a sufficient stage of development that the test case could be implemented inside the full ORIGAME environment and run for demonstrative and comparative purposes. The code for the test case was identical to that developed in Section 3.2.6, except that Python 2.6 does not support ordered dictionaries. As a result, this test case was eliminated here. Within the word count test functions, as described in Section 2.3, the following data types were used:

- Arrays (dynamic), noting that two separate arrays were used;
- Class and array; and,
- Dictionary.

In addition, the ORIGAME environment was ported without its GUI to run in Python 2.7. As a result, the ordered dictionary data type was added back into the test case and tested again.

3.3.3 SimEvents

The first development environment examined was an extension of Matlab, known as SimuLink. SimuLink is a simulation environment for dynamic systems, with a DES toolbox called SimEvents. SimEvents contains a DES engine and component libraries for managing and processing asynchronous events.

As with SimPy, the test case was adapted it to fit within the DES construct. The details of the adaptations within the SimEvents DES and SimuLink visual programming environment are provided in Annex A.

Within the word count test functions, as described in Section 2.3, the following data types were used:

- Structure and array (dynamic);
- Class and array (dynamic);
- Map; and,
- Cell array.

The major drawback of the use of SimEvents is that it requires a full license of Matlab, SimuLink, and the SimEvents toolbox to run. A standalone executable version cannot be created, making deployment in a parallel processing environment prohibitive unless the Matlab distributed computing server license is purchased for computing nodes.

3.3.4 Simio

Simio is built on an object-oriented concept, and comes with a set of standard modeling objects that the user can "click and drag" in two- or three-dimensional space to create models. These objects can be modified by the user or used as building blocks to create other objects as needed. Simio's greatest strength lies in its graphics engine, which allows the user to quickly visualize and reproduce physical processes with ease.

Simio also allows the user to create system extensions in the form of DLL's, where standard programming languages can be used to create these libraries. For the purposes of the test case, a user extension was written using VC#.NET. Although the VC#.NET code was migrated to the DLL's with minimal changes, getting the code to be executed by Simio properly required some trial and error because the User Extension documentation was found to be lacking. Further details of the test case adaptation to the Simio visual programming environment can be found in Annex A.

Within the word count test functions, as described in Section 2.3, the following data types were used:

- Structure and array (dynamic);
- Class and list; and,
- Dictionary.

Due to the fact that some data structure combinations were not explored until after the time-limited trial license expired, the structure/list and class/array test functions were not evaluated.

3.3.5 Simul8

Like Simio, Simul8 is specifically designed for DES and is not a general programming language. Also like Simio, it has a “click and drag” visual programming interface and a base set of simulation construction objects. However, with Simul8, these are the only objects that can be used; new objects cannot be built. Object properties are limited and the quality of the graphical user interface leaves much to be desired (for more information refer to [1]).

Two programming extension options are provided. First, extensions can be written in Visual Basic for Applications (VBA). Though offered in the version tested here, this interface is no longer supported by the manufacturer. The majority of the VB6 code from the test case could have been copied into an Excel workbook and executed as a macro through Simul8, but would have provided performance no better (and likely worse, since macros are not compiled code) than VB6. However, without further support of the manufacturer, the VBA extension did not seem worthwhile to explore.

Instead, the primary programming extension is a proprietary language called Visual Logic. The Visual Logic programming interface is cumbersome, as commands must be selected and entered from a GUI list to the left of the programming area. 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.

Visual Logic code can be written in standalone segments or it can be associated with objects in the simulation. The association to objects is performed through dialogs handling an object's properties; however, the association must be set up before any coding is attempted. 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 functions in the test program could not be implemented. The string loading procedure could not be readily constructed either. With time, some of these functions could likely have been fabricated from Visual Logic but, with only a 14 day trial evaluation version, it was not possible.

Since the time of testing in November 2011, a new version of Simul8 was released. According to the manufacturer’s website documentation, significant changes were made, bringing the capability of the software more in line with what Simio is capable of. Due to time constraints, no testing was performed on the new version.

3.3.6 AnyLogic

AnyLogic was ruled out as a possible software package after consultation with the DRDC CORA Information Technology team, as the development license requires drivers that are not acceptable

for security reasons on the DRENet computer network. Although the run-time version could be installed, restricting Tyche development and debugging to standalone computers was deemed a sufficient complication to remove AnyLogic from further consideration.

4 System configuration

4.1 Hardware

Initially, it was hoped that all test cases could be run on the same hardware to ensure a fair comparison between development environments. However, due to time constraints, division of labour, and location of specific software licenses, three separate systems were used with overlapping test cases to provide a basis for comparison. System 1, shown in Table 1, was provided by Terry Restoule (Contractor). Systems 2 and 3 were provided by DRDC.

Table 1: Hardware systems used for testing, including primary specifications.

Hardware	Specifications		
	System 1	**System 2**	**System 3**
Style	Desktop	Laptop	Laptop
Make and Model	Acer Aspire M3910	HP Elitebook 2540p	HP Elitebook 8540p
RAM	6 GB	4 GB	4 GB
CPU Type	Intel Core i5	Intel Core i7	Intel Core i5
CPU Model	i5-650	L640	i5-520M
CPU Speed	3.20 GHz	2.13 GHz	2.40 GHz
Number of Cores[5]	2	2	2
Additional Features	Hyperthreading and Turbo Boost	Hyperthreading and Turbo Boost	Hyperthreading and Turbo Boost

4.2 Software

All test cases were run under the 64-bit version of Microsoft Windows 7 (Enterprise or Home Premium Edition) with Service Pack 1 (SP1). The software listed in Table 2 was then installed for code development (and compilation, where possible).

[5] Although each machine was equipped with moiré than one CPU core, the test cases were designed as single-threaded applications that ran on only one core at a time. The exception to this may be Simio, where multi-threading/parallel processing is claimed to be used by default; however, no evidence favouring either case was obtained during the runs.

Table 2: Development environments used for test case development.

Software	Source
Microsoft Visual Basic 6.0 SP6	Purchased, standalone license from DRDC CORA
Microsoft Visual Basic.NET 2010 Express	http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-basic-express
Microsoft Visual C#.NET 2010 Express	http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express
Microsoft Visual C++.NET 2010 Express	http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express
DEV C++ Version 4.9.9.2	http://www.bloodshed.net/devcpp.html
Python 2.7.2, using: SimPy 2.1.0	http://www.python.org/getit/releases/2.7/ http://sourceforge.net/projects/simpy/files/simpy/SimPy-2.2/
ORIGAME, using: Python 2.6 and Panada3D SDK 1.7.2	stephen.okazawa@forces.gc.ca http://www.panda3d.org/download.php
Matlab R2010a with SimuLink, the SimEvents toolbox and the Matlab Compiler, using: Microsoft Visual C++ SP1 in Visual Studio 8 and Matlab Compiler Runtime (MCR) 7.13	Purchased, network license from DRDC CORA Purchased, standalone license from DRDC CORA MCR included with above Matlab network licenses
Simio Release 4 (30-day trial of Design Professional version)	http://www.simio.com/evaluate.php After contact with sales representative, upgrade from Evaluation (unlimited time, limited modelling and no DLL linking capabilities) to Design Professional (full version)
Simul8 2010 (14-day trial of Professional version)	http://www.simul8.com/support/eval/index.php

5 Test case results and discussion

5.1 Computational performance

As mentioned in Section 4.1, not all implementations of the test case could be run on a single machine. License and time limitations restricted the use of Simio to System 1, Matlab (interpreted m-file) and SimEvents to System 2, and the original prototype version of ORIGAME[6] on System 3. Since System 3 was only used to compare Python and ORIGAME performance, none of the other test cases were run on it. The results for System 1 are presented in Table 3, System 2 in Table 4, and System 3 in Table 5. Average function times are given in seconds, while total simulation run times are given in minutes:seconds.

It is important to note that not all test case implementations contained the same number or type of functions for the book read and data manipulation (word count) application. Consequently, all results must be viewed to compare average run times for similar functions. The total run time is presented as an indicator of order of magnitude of performance, as well as the additional program overhead above the time to perform individual functions. The calculation of total overhead time is presented in Table 6 for System 2, which is generally representative of the range of test cases.

The comparative analysis of the computational performance testing will be discussed in the following subsections.

[6] Notably, the original ORIGAME framework was developed using Python 2.6, rather than Python 2.7. Python 2.6 does not contain the ordered dictionary, which is reflected in the results shown in Table 5. The test case was run in the original ORIGAME and a modified version (minus the graphical user interface) using Python 2.7.

Table 3: Test case results on System 1.

Development Environment	Average Function Timing (seconds)									Total Run Time (minutes: seconds)
			Book Read and Data Manipulation							
	String Manipulation	Arithmetic Operations	Type Definition/Structure and Array	Structure and List/Vector	Class and Collection/List/Vector	Class and Dynamic Array	Dictionary/Map	Ordered Dictionary	Cell Array	
VB6	0.016	0.014	2.660		10.273	23.330	0.178			6:02.858
VB.NET	0.037	0.010	1.114		1.191	1.793	0.058			0:41.309
VC#.NET	0.040	0.006	0.435	0.432	0.423	0.655	0.053			0:20.592
VC++.NET	0.042	0.008	5.715	6.483	43.485	44.697	1.703			16:56.502
DEV C++	0.019	0.000	1.256	1.331	3.760	3.577	0.187			1:41.587
Python 2.7	0.014	0.117	1.151			21.534	0.032	0.117		3:49.969
SimPy	0.014	0.117	1.139			21.316	0.032	0.117		3:47.648
ORIGAME (Python 2.7)	0.014	0.129	1.157			21.525	0.032	0.117		3:50.129
Matlab (EXE, Partially Optimized)	0.007	0.586	1.380			1.775	1.418		1.157	1:38.249
Simio	0.061	0.005	0.487			0.685	0.052			>0:12.884*

* Note that for the Simio test case, the exact run time was not available. When running the Simio tests, the entities did not always stay in the server long enough to completely append their results to the output file. The numbers in the Simio row are taken from 10 recorded iterations made during several different runs of the model.

Table 4: Test case results on System 2.

Development Environment	Average Function Timing (seconds)									Total Run Time (minutes: seconds)
	String Manipulation	Arithmetic Operations	Book Read and Data Manipulation							
			Type Definition/ Structure and Array	Structure and List/Vector	Class and Collection/List/Vector	Class and Dynamic Array	Dictionary/Map	Ordered Dictionary	Cell Array	
VB6	0.019	0.016	3.345		12.167	29.037	0.225			7:28.830
VB.NET	0.064	0.009	1.390		1.363	2.218	0.061			0:51.278
VC#.NET	0.047	0.013	0.516	0.515	0.506	0.779	0.062			0:25.475
VC++.NET	0.051	0.011	6.730	7.490	49.272	51.374	1.268			19:23.175
DEV C++	0.023	0.008	1.509	1.600	3.974	3.962	0.223			1:53.402
Python 2.7	0.014	0.127	1.243			22.413	0.035	0.125		3:57.399
SimPy	0.014	0.126	1.248			21.905	0.035	0.128		3:54.887
ORIGAME (Python 2.7)	0.014	0.140	1.252			21.831	0.034	0.125		3:54.386
Matlab (Interpreted, Not Optimized)	0.029	6.477	48.460			306.577	19.500		195.868	96:45.946
Matlab (Interpreted, Partially Optimized)	0.041	0.654	1.653			2.242	1.705		1.401	1:50.859
Matlab (EXE, Partially Optimized)	0.013	0.643	1.598			2.377	1.657		1.361	1:47.962
SimEvents	0.001	0.940	1.637			2.181	1.682		1.392	1:56.826

Table 5: Test case results on System 3.

Development Environment	Average Function Timing (seconds)									Total Run Time (minutes: seconds)
	String Manipulation	Arithmetic Operations	Book Read and Data Manipulation							
			Type Definition/ Structure and Array	Structure and List/Vector	Class and Collection/List/Vector	Class and Dynamic Array	Dictionary/Map	Ordered Dictionary	Cell Array	
ORIGAME (Python 2.6)	0.014	0.072	1.256			16.925	0.033			3:02.301
ORIGAME (Python 2.7)	0.018	0.074	1.551			22.015	0.040	0.157		3:58.270
Python 2.7	0.013	0.093	1.340			17.049	0.031	0.111		3:06.677

Table 6: Test case overhead on System 2.

Development Environment	Total Run Time (seconds)	Total Function Time (seconds)	Overhead Time (seconds)	Percentage Overhead
VB6	448.830	448.066	0.764	0.170%
VB.NET	51.278	51.059	0.219	0.427%
VC#.NET	25.475	24.369	1.106	4.342%
VC++.NET	1163.175	1161.967	1.208	0.104%
DEV C++	113.402	112.986	0.416	0.367%
Python 2.7	237.399	239.571	-2.172	-0.915%*
SimPy	234.887	234.553	0.334	0.142%
ORIGAME (Python 2.7)	234.386	233.966	0.420	0.179%
Matlab (Interpreted, Not Optimized)	5805.946	5769.111	36.835	0.634%
Matlab (Interpreted, Partially Optimized)	110.859	76.957	33.902	30.6%
Matlab (EXE, Partially Optimized)	107.962	76.487	31.475	29.2%
SimEvents	116.826	78.338	38.488	32.9%

* It is not known why the total clock run time is apparently slower than the sum of the individual function run times in this case.

5.1.1 System comparison

When looking at run times across the range of development environments and functions between computer systems, System 1 is able to execute most of the test case faster than System 2 or 3 (with the exception of Python 2.7, which will be discussed further in Section 5.1.2). This would seem to support the argument that computational performance scales with CPU speed, similar to the way Tyche performs currently [9]. Insufficient data exists to determine the scaling relationship (and it is not within the scope of this study).

5.1.2 Development environment comparison

The two fastest development environments to run the test case as a whole[7] are VC#.NET and VB.NET. VC#.NET is twice as fast as VB.NET, while running a total of seven different functions, versus six in VB.NET. Matlab runs in third place, with DEV C++ and SimEvents not far behind. The executable version of Matlab runs slightly faster than the interpreted version, but does not give as much improvement as one might expect with compiled code. DEV C++ is running seven functions versus Matlab's six; however, the lack of familiarity with the Matlab functions and limited development time meant that the code was not fully optimized[8] for the Matlab development environment. The exact performance ranking of Matlab and DEV C++ is not important though, as both ran more than two times slower than VB.NET and more than eight times slower than VC#.NET. In an application such as Tyche for MCDES, speed is of the utmost priority.

Individual function run times in Simio were extremely fast; as one would expect when running the VC#.NET code to perform the test functions. However, with the test case as developed, there was no way to accurately estimate total run time[9] and determine the overhead of running the program inside the Simio development environment. Although it may be possible to resolve the issue (entities not staying in the server long enough to completely append results to the output file) utilizing a similar method to the SimEvents procedure (refer to Annex A.1), expiration of the evaluation license prevented further testing.

As anticipated, the interpreted version of the test case in variations of Python and SimPy were significantly slower than VB.NET or VC#.NET. In addition, inconsistencies were displayed between versions and various add-ons. The Python 2.7 version did not seem to scale with CPU speed and the ORIGAME (Python 2.7) version was not consistently slower (or faster) than the plain Python 2.7 version. Since the results produced on System 3 in Table 5 were primarily to illustrate the difference between the test case within the ORIGAME framework and plain Python, no conclusions, unfortunately, about the overhead involved can be drawn at this time.

---

[7] Applications with significantly different proportions of computational types would, of course, fare differently.

[8] A review by a Matlab user recommended that the following changes be made to improve test case performance: cells and string be pre-allocated (as per http://blinkdagger.com/matlab/matlab-speed-up-your-code-by-preallocating-the-size-of-arrays-cells-and-structures/); anonymous functions should be replaced with explicit function definitions; and replace `fprintf()` with `dlmwrite()`, `export()` or `save()`.

[9] The total run time estimated in Table 3 is simply a summation of the individual function run times from 10 separate, non-contiguous iterations. Only five test functions were included (compared to six or seven in the other development environments).

VB6 was one of the slowest test case runs, with only VC++.NET and the non-optimized version of Matlab performing slower.

5.1.3 Data type comparison

Within the book read and data manipulation function, various data types are used to store the words and their counts. The comparison between these data types can provide a recommendation for the fastest implementations within the full Tyche simulation engine.

First, it is necessary to remove from consideration the data types that do not maintain order of entry. The VC++.Net and DEV C++ map, the Python dictionary, and the Matlab map are excluded from further discussion here. The Matlab test case version that is not optimized is also removed from consideration due to extremely poor performance.

The fastest performing data types are the VB6, VB.NET and VC#.NET dictionaries, and the Python ordered dictionary. These data types are often at least an order of magnitude faster than any other comparable type in the same development environment. Matlab cell arrays are also the best performer in that environment, but not by a significant amount over the structure and array combination. If the data to be represented fits within the simple key and value pair structure (of value types[10]), then these are by far the best choice.

The next best combination of data types are type definitions/structures and arrays. The structure and list/vector combination (where possible) performs similarly. Interestingly, in VC#.NET, the class and array, or class and list combinations are also quite fast. In all other development environments, the introduction of the class data type increases run time significantly.

DEV C++ is the only environment in which the class and array combination is slower than the class and collection/list/vector combination. In general, it seems that the more complex reference-type objects require greater processing time (as these variables are accessed through the heap, rather than the stack, with additional overhead to support greater complexity).

5.1.4 Overhead comparison

When the amount of time the test case spends in tasks over and above the functions defined in Sections 2.3.1 to 2.3.3 is examined, it is for the purpose of estimating the order of magnitude for the overhead. No attempt is made to determine if the overhead scales with number of functions, function run time, or how it scales between computer systems. The purpose here is to determine if the overhead forms a significant portion of the computational time.

The calculation of total overhead time was presented in Table 6 for System 2, covering the range of development environments of interest with one exception: Simio. Simio was not run on System 2 and, as stated in Table 3, Simio was not able to run the entire test case from start to finish. A total run time could not be calculated and, as a result, it is not possible to determine the overhead associated with running the test case in the Simio software package. This further indicates issues

[10] Dictionaries can certainly hold reference types as well, but the performance suffers. Refer to Annex B for more information.

with implementing user-defined extensions, a fundamental requirement in this development environment for the implementation of Tyche.

For the remainder of the development environments, the majority of the overhead is less than two seconds. The exception here is Matlab and SimEvents. In all variations of the test case implemented through Matlab, there are more than 30 seconds of overhead computation. This also appears to be independent of total run time and interpreted/executable version, in the limited testing performed here.

Since the test case is a fairly small program with few iterations to compute, the overhead appears to be a significant portion of the total run time in some development environments. However, as the program complexity, data requirements, and iterations computed increase, the total simulation run time will increase – rendering the overhead less significant in the overall performance drivers.

5.2 Discussion

While computational speed to reduce simulation run time is the primary factor driving the redevelopment of Tyche, there are several other factors that must be considered as well. Not only should the selected development environment be able to support the features and computations currently performed in Tyche, but the ease of code conversion, maintenance, debugging, and quality of help support should also be taken into account. The number, type, and cost of licenses required to develop code and run the final product in a parallel processing environment are also a concern. Table 7 summarizes the advantages and disadvantages of each of the development environments tested.

Table 7: Qualitative assessment of development environments for implementation of Tyche.

Development Environment	**Advantages**	**Disadvantages**
VB.NET	+ Similarity to current code for greatest ease of conversion. + A well-developed (free) IDE with comprehensive debugging tools and help support. + As part of the Microsoft Visual Studio package and the .NET framework, the programming language should be supported for some time to come.	- Moderate computational performance.

Development Environment	Advantages	Disadvantages
VC#.NET	+ Best computational performance. + Some similarity to VB for ease of code conversion. + A well-developed (free) IDE with comprehensive debugging tools and help support. + As part of the Microsoft Visual Studio package and the .NET framework, the programming language should be supported for some time to come.	- Strongly typed language requires programmer to write additional code and maintain more syntactical structures than either version of VB.
VC++.NET	+ A well-developed (free) IDE with comprehensive debugging tools and help support. + As part of the Microsoft Visual Studio package and the .NET framework, the programming language should be supported for some time to come.	- Extremely poor computational performance. - Suffers from functional clash between managed and unmanaged code. Potential issues may arise with code maintenance. - Code conversion requires significant effort.
Dev C++	+ Well-developed (free) IDE with debugging tools. + Many open source libraries available. + Open source nature and wide usage of C++ useful for code maintenance.	- Moderate computational performance. - Code conversion requires significant effort.

Development Environment	Advantages	Disadvantages
Python	+ Easy to write and run code, but some effort required for conversion from VB6. + Many open source libraries available.	- Poor computational performance. - Lack of versioning continuity and agreement about which version to use. Python 3.2.2 is the most recent version, but older versions (such as Python 2.7 used here) are still in common use – depending on the preference of the user. - Difficult to determine (without input from experienced user) the exact combination of add-on site packages that work together correctly - Documentation is difficult to understand. Newer versions do not always have sufficiently worked examples to be useful. - Many IDE's to choose from, with varying levels of debugging support.
SimPy	+ Easy to write code, but some effort required for conversion from VB6. + Easy to understand, as it is a DES organized inside a serialized function construct. + Well documented help and examples. + Open source, and can include any Python library.	- Poor computational performance.
Matlab	+ Potential to significantly improve computational performance if code is fully optimized for matrix math. + Easy to write code, but some effort required for conversion from VB6. + Well-developed IDE with comprehensive debugging tools and help support. + Long history of program development and easy access to code for maintenance activities.	- Requires licenses (cost $). Cost will scale with number of CPU's to perform computing in a high-performance computing system. - Moderate computational performance. - Requires complete restructure of Tyche simulation engine to utilize matrix math. - Lose ability to easily trace data (using structure or class objects) when debugging matrix math.

Development Environment	Advantages	Disadvantages
SimEvents	+ Potential to improve computational performance if code is significantly restructured. + Well-developed IDE with debugging tools and help support. + Long history of program development and easy access to code for maintenance activities.	- Requires additional toolboxes/licenses (cost $) above basic Matlab license. - Moderate computational performance. - Limitations to modelling imposed through SimuLink. - Requires complete restructure of Tyche simulation engine and significant effort for code development (visual programming interface as well as Matlab code syntax).
Simio	+ Exhibits greatest potential to employ unique features of DES software packages and speed of best programming languages.	- Requires licenses (cost $). - Unknown computational overhead. - Requires complete restructure of Tyche simulation engine. - Programmers must learn and maintain skill in visual programming interface, as well as a separate programming language for user-defined extensions (DLL's). - Issues with documentation for user-defined extensions.
Simul8	+	- Requires licenses (cost $). - Removed from consideration for not meeting minimum functional requirements.
Anylogic	+	- Requires licenses (cost $). - Removed from consideration for not meeting minimum functional requirements.

When the computational performance and qualitative impressions of each development environment are combined, the best choice for Tyche is VC#.NET. It is the fastest performing code on the test case developed herein and has the best suite of supporting features. It also has relatively few disadvantages when trying to minimize both cost and effort for the redevelopment project.

6 Conclusion

When the computational performance and qualitative impressions of each development environment are combined, the best choice for Tyche is VC#.NET. It is the fastest performing code on the test case developed herein and has the best suite of supporting features. Code conversion will take time, but would not seem to be as much of an issue as some of the other development environments. Code debugging and maintenance will be relatively straightforward. The VC#.NET IDE provides comprehensive debugging tools and help support. Since the 2010 VC#.NET Express version from Microsoft is free to download, there is no additional cost to the Tyche ARP to write code or run executable files. All of the .NET environments provide support for parallel processing using MS MPI, ensuring future programming needs for operation in a high performance computing environment will be met.

During the time it took to produce this report, much of the Tyche code conversion from VB6 to VC#.NET took place. Based on the poor performance of the class objects noted in Section 5.1, structures were used as much as possible. However, due to the complexity of the data contained within the original classes, it was necessary to test further combinations of structures with arrays, dictionaries, and lists than those given in Section 3.2.3. Annex B details the additional analysis using modifications of the test case from Section 2.3.

Until code conversion is completed and debugged, it is not known exactly how much the new programming language will affect the performance of the program. The VC#.NET test case showed a 96% reduction in run time compared to VB6. Extrapolating to a full simulation run on the FMS II data set [8, 9], individual simulation runs that previously took 3 hours to run could be reduced to less than 8 minutes. Although the same scaling is not likely, a significant improvement in performance is still expected. Once the Tyche simulation engine code has been fully rewritten and debugged, a performance comparison with the current version (Tyche 2.3.4) is planned.

This page intentionally left blank.

References

[1] Restoule, T. (2012), Notes on the simultest programs used in the language selection process: Tyche 3.0 simulation engine project, (DRDC CORA CR 2012-XXX), LeverageTek IT Solutions, Ottawa, ON.

[2] Eisler, C., Allen, D., Blakeney, D., Burton, R.M.H., and MacDonald, G. (2007), Tyche (version 1.0): Software architecture and design documentation, (DRDC CORA TR 2007-05), Defence R&D Canada - CORA.

[3] Allen, D., Blakeney, D., Burton, R.M.H., and Purcell, D., LCdr (2005), Fleet mix study: Determining the capacity and capability of the future naval force structure, (DRDC CORA TR 2005-38), Defence R&D Canada - CORA.

[4] Allen, D., Eisler, C., and Forget, A. (2006), A user guide to tyche version 2.0: Providing a joint flavour to Tyche, (DRDC CORA TR 2006-14), Defence R&D Canada - CORA.

[5] Eisler, C. (2007), Fleet mix study: Calculating sealift supply and demand for iteration II, (DRDC CORA TN 2007-011), Defence R&D Canada - CORA.

[6] Heppenstall, D. (2007), A user's guide and programmer's manual to tyche version 2.2: Handling simulations with greater efficiency and flexibility, (DRDC CORA CR 2007-01), University of Guelph, Guelph, ON.

[7] Michalowski, P. (2009), Tyche 2.3: Supporting documentation, (DRDC CORA CR 2009-005), Brainhunter, Inc., Ottawa, On.

[8] Bourque, A., and Eisler, C. (2010), Fleet mix study iteration II: Making the case for the capacity of the "Navy After Next", (DRDC CORA TR 2010-159), Defence R&D Canada - CORA.

[9] Eisler, C. (2010), Benchmark testing using tyche: From Personal Computer to Desktop Supercomputer, (DRDC CORA TN 2010-251), Defence R&D Canada - CORA.

[10] 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.

[11] Eisler, C., Allen, D., Forget, A., Heppenstall, D., and Michalowski, P. (2009), Tyche 2.3 help files. DRDC CORA.

[12] Keserovic, S., Mortenson, D., and Nathan, A. (2003), An overview of managed/unmanaged code interoperability (online), Microsoft Corporation, http://msdn.microsoft.com/en-us/library/ms973872.aspx (Access Date: 09/05/2012).

[13] Bouayed, Z., Moorhead, P., Séguin, R., and Okazawa, S. (2011), R4 HR TDP concept of use: From a stove-piped approach to an integrated approach, (DRDC DGMPRA 2011-011), Defence R&D Canada - DGMPRA.

Annex A Visual programming models

A.1 SimEvents

The SimEvents model is based on *entities* that are processed by *servers*. To represent the test case, ten entities (iterations) must be generated and processed by a single server. To create these entities, an event-based sequence is used to output ten, nominally small inter-creation times to the time-based entity generator (source) as shown at the left of Figure A-3. These entities flow to a single server (first in, first out), while a count of total entities is output to a discrete event subsystem. The scopes shown in Figure A-3 are used only for debugging purposes and have no effect on the system.

The first single server has no processing delay time, and serves as a holding queue for all of the entities. The entities are released when the output port is available, flowing to the second single server (also first in, first out) in the sequence. This server does employ a processing delay, input from the discrete event subsystem. Once each entity is processed, they flow to the entity sink at the right of Figure A-3.

Key to the model's operation is the discrete event subsystem. At every change in input (i.e., when an entity is released from the first server), the subsystem shown in Figure A-4 is fired. Each time a rise in input signal is detected, an embedded Matlab function (*__fcn_ss__* in Figure A-4) is called. The embedded Matlab function contains the function definitions and calls to the test case functions described in Section 2.3.

The total time for processing the embedded Matlab function is used as output back to the main system to ensure that the simulation time is synched with the computation time. Each "iteration" entity is then processed by the server according to the time it takes to complete the necessary computations and write the data to the output file. This peculiar modelling is necessary to calculate the total simulation run time. If it were not used, the simulation clock would not count the time required to process the embedded Matlab function.

Because the embedded Matlab function is repeated every iteration, the call to zip the output file could not be included within it. Due to time constraints, that function was omitted (although proven for use in the Matlab version of the test case). Likely, a second discrete event subsystem would be necessary, with another embedded Matlab function for zipping the output file. This could be inserted in the system to be triggered upon completion of processing of all entities in the second server.

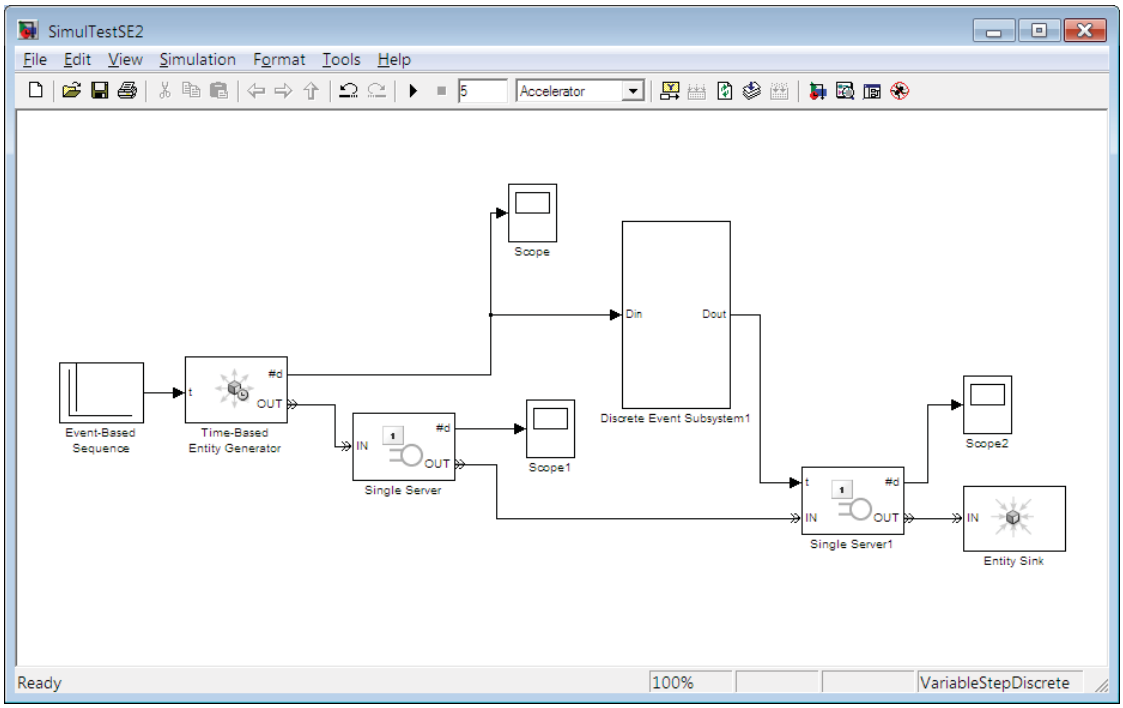

Figure A-3: The SimEvents test case visual model.

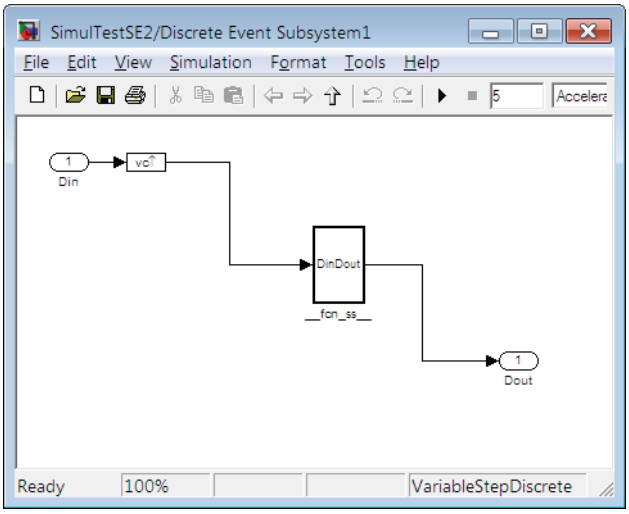

Figure A-4: The SimEvents discrete event subsystem.

A.2 Simio

Simio operates similarly to SimEvents, with entities, servers, sources and sinks (among others). The primary difference is that Simio is event driven – it is not designed as a continuous-time signal processing system like SimuLink. As a result, the model is significantly simplified when compared to SimEvents. Figure A-5 illustrates the source-server-sink model in the Simio facility diagram used to represent the test case.

The source generates entities (iterations) when the output flow is empty, to a maximum of ten. The output from the source is linked to the input of the server, which processes one entity at a time (first in, first out). The sink accepted the entities once the server is finished processing them.

To model the delay introduced when processing an entity, a process was created. The process was triggered every time an entity entered the server, and called a user-defined system extension in the form of a DLL. The DLL was written in VC#.NET to perform the test case functions for a single iteration and placed in the Simio *UserExtensions* directory. Simio automatically detects extensions in this location and makes them available for use in the process diagram. A second extension was written to zip the output file and executed through Simio as a standard process using an "on run ending" event.

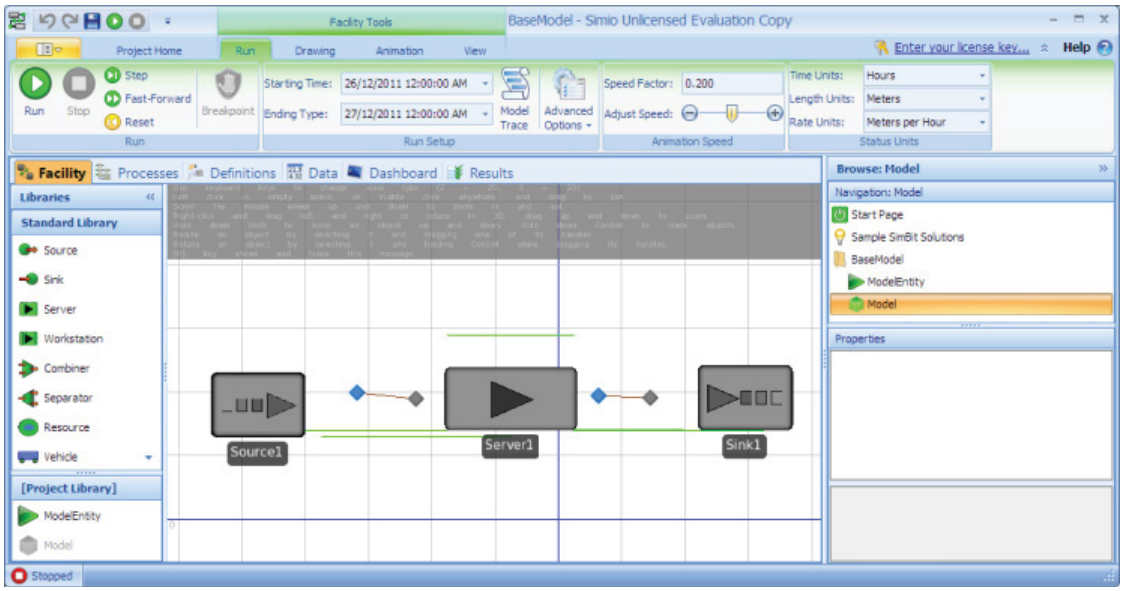

Figure A-5: The Simio test case visual model.

This page intentionally left blank.

Annex B Additional VC#.NET results

During the publication of this report, development work on Tyche 3.0 started. Many of the data structures were migrated to simple inherent data types, structures or dictionaries. However, some of the existing Tyche data classes, arrays of classes, multidimensional arrays, and lists of classes needed to be converted to simpler, faster data types. As a result, some additional testing of the book read and data manipulation test case was conducted to determine the performance of the following combinations:

- A dictionary, where the key is an integer and the value is a structure (integer and string);
- A dictionary, where the key is a string and the value is a structure (integer and string);
- A dictionary, where the key is an integer and the value is a class;
- A dictionary, where the key is a string and the value is a class;
- A jagged array (an array of arrays), where the secondary array elements are lists of structures; and,
- An array of structures, whose elements are secondary structures.

The VC#.NET test case was modified to include these combinations, and the results (run on System 2) are presented in Table A-8.

Table A-8: Test case in VC#.NET with additional combinations.

Book Read and Data Manipulation	**Average Function Timing (seconds)**
String Manipulation	0.055
Arithmetic Operations	0.005
Array of Structures [Structure(Integer; String)]	0.518
List of Structures < Structure(Integer; String)>	0.511
List of Classes < Class(Integer; String)>	0.493
Array of Classes [Class(Integer; String)]	0.771
Dictionary (Integer, String)	0.055
Dictionary (Integer, Structure(Integer, String))	1.682
Dictionary (String, Structure(Integer, String))	1.144
Dictionary (Integer, Class(Integer; String))	1.375
Dictionary (String, Class(Integer; String))	1.512
Array of Structures of Structures [(Integer, Structure(Integer, String))]	0.529
Jagged Array [[<Structure(Integer, String)>]]	0.576

Of note, when the dictionaries are expanded to include more complex data types, their performance decreases dramatically. Since much of the data is linked through classes of classes (now converted to structures of structures), dictionaries are not recommended as containers for said objects. Lists and arrays perform similarly, both approximately three times faster than the dictionary counterparts. The array of structures, whose elements are another structure type, were used to confirm that the performance does not suffer significantly when nested structures are used for linked data.

The jagged array is used in place of a two-dimensional array (from VB6), and shows similar performance (based on similar data content) to the lists and arrays of structures. This allows the inner and outer arrays to be resized using *Array.Resize*, as multidimensional arrays in VC#.NET have no built-in support for dynamic resizing.

List of symbols/abbreviations/acronyms/initialisms

ARP	Applied Research Project
ASCII	American Standard Code for Information Interchange
CLR	Common Language Runtime
CORA	Centre for Operational Research and Analysis
CPU	Central Processing Unit
DES	Discrete Event Simulation
DGMPRA	Director General Military Personnel Research and Analysis
DLL	Dynamically Linked Library
DND	Department of National Defence
DRDC	Defence Research & Development Canada
DRDKIM	Director Research and Development Knowledge and Information Management
EXE	Executable
FMS	Fleet Mix Study
IDE	Integrated Development Environment
MCDES	Monte Carlo Discrete Event Simulation
MCR	Matlab Compiler Runtime
MS MPI	Microsoft Message Passing Interface
OrderedDict	Ordered Dictionary
R4 HR TDP	Right Person, Right Qualifications, Right Place, Right Time Human Resources Technology Demonstration Project
RAM	Random Access Memory
SP1	Service Pack 1
Struct	Structure
VB6	Visual Basic 6.0
VBA	Visual Basic for Applications
VC#.NET	Visual C#.NET
VC++.NET	Visual C++.NET

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.)

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

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.)

Tyche 3.0 Development: Comparison of Development Environments for a Monte Carlo Discrete Event Simulation (MCDES)

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

Eisler, C;

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.)
September 2012	60	13

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.)

Technical Memorandum

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.)
ARP 11ic	

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 TM 2012-231	

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.)

Tyche is a Monte Carlo discrete event simulation for analysis of force structures. To date, Tyche has been implemented in Microsoft Visual Basic 6. Due to the fact that Microsoft no longer supports this version of Visual Basic, as well as the need to increase both simulation speed and capability, an Applied Research Project (ARP 11ic) was begun to rewrite the Tyche tool in a new programming language and build upon its functionality. The first phase of the project was to identify and select a suitable programming language or software package that could support the functionality of the current program, as well as future development for parallel processing in a high-performance computing environment. After a survey of existing programs and open literature, six potential development environments were identified with the highest potential to meet the functional requirements necessary to support Tyche: Microsoft Visual Studio.NET, Simio, AnyLogic, Simul8, SimEvents (Matlab), and SimPy (Python). To determine the best development environment, an iterative test program was created to compare the speed and ease of implementation on the kinds of computations that Tyche performs. Results indicated that Microsoft Visual C#.NET is the fastest for computation of the types calculations Tyche performs, as well as the most flexible and easy to use development environment (given the history of Tyche and its developers). A variety of data types and structures were also compared for performance to make recommendations for future implementation.

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.)

Speed; Programming Language; Discrete Event Simulation; Comparison;

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

www.drdc-rddc.gc.ca