Analysis of the Tyche Simulation Engine and Recommendations for Future Development

Ruibiao Jaff Guo
CAE Professional Services

Jeremy Brooks
CAE Professional Services

DRDC CORA CR 2012–081
April 2012

Defence R&D Canada
Centre for Operational Research and Analysis

Maritime Operational Research Team
Director of Maritime Strategy

National Defence Défense nationale

Canada

Analysis of the Tyche Simulation Engine and Recommendations for Future Development

Ruibiao Jaff Guo
CAE Professional Services

Jeremy Brooks
CAE Professional Services

Prepared By:
CAE Professional Services
300-1135 Innovation Drive
Ottawa, ON K2K 3G7 CANADA
CAE Professional Services
Contractor's Document Number: CAE PS #5160-016 Version 01
Contract Project Manager: Richard Percival, 613-314-6449
PWGSC Contract Number: W7714-4500825828
CSA: François-Alex Bourque, Defence Scientist, 613-992-3206

Defence R&D Canada – CORA

Contract Report
DRDC CORA CR 2012-081
April 2012

Principal Author

Original signed by Ruibiao Jaff Guo

Ruibiao Jaff Guo

Cognitive Modelling Scientist

Approved by

Original signed by Dr. R. E. Mitchell

Dr. R. E. Mitchell

Head Maritime and ISR Systems OR

Approved for release by

Original signed by Paul Comeau

Paul Comeau

Chief Scientist

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

Abstract

The objective of this project was to analyze the current Tyche simulation system, identify the problems that affect simulation execution time, and make recommendations to speed up Tyche simulation execution.

The analysis of the Tyche system consisted of identifying physical, conceptual and dynamic entities/components as well as their relationships, and extracting function flowcharts in the simulation process to help understand the system and diagnose problems.

Based on the results of the Tyche system analysis, two kinds of problems were identified that affect simulation execution time including the problem of single-thread execution and the issues of redundant or repeated operations in simulation iterations.

Three categories of recommendations are proposed for speeding up simulation execution and future Tyche development, comprising a parallel system architecture, a group of approaches for optimizing the design of the simulation engine, and a set of alternative programming environments for future Tyche development.

Résumé

Le projet avait pour but d'analyser le système de simulation Tyche actuel, de cerner les problèmes qui nuisent à la vitesse d'exécution et de formuler des recommandations en vue d'accélérer l'exécution de simulations Tyche.

L'analyse du système Tyche comportait l'identification des composants et des entités physiques, conceptuelles et dynamiques, ainsi que des relations entre ceux-ci, et l'extraction d'organigrammes des fonctions du processus de simulation pour permettre de mieux comprendre le système et de diagnostiquer les problèmes.

Les résultats de l'analyse du système Tyche ont permis de cerner deux types de problèmes qui nuisent à la vitesse d'exécution de simulations : utilisation d'un seul fil d'exécution et opérations redondantes ou répétées au cours des itérations de simulation.

Nous proposons trois catégories de recommandations pour accélérer les simulations et guider le développement futur : une architecture système parallèle, un ensemble de démarches pour optimiser la conception du moteur de simulation et un ensemble d'autres environnements de programmation pour le développement futur de Tyche.

This page intentionally left blank.

Executive summary

Analysis of the Tyche Simulation Engine and Recommendations for Future Development

R. J. Guo; J. Brooks; DRDC CORA CR 2012-081; Defence R&D Canada – CORA; April 2012.

Introduction: The Tyche simulation system is a Monte Carlo based discrete event simulation system developed by Defence Research and Development Canada's (DRDC) Center for Operational Research and Analysis (CORA) for conducting force structure analysis at the level of fleets in support of strategic planning.

With the current Tyche software system, a typical force structure simulation takes several hours to complete on a single-processor platform. A current maintenance priority is to shorten simulation runtime to increase the capacity to test a large number of fleet options in the minimal amount of time. Specifically, redesigning the architecture to support parallel processing in a cluster/grid environment is sought as CORA will acquire a high-performance computing environment. CAE Professional Services (Canada) Inc. (CAEPS) was contracted to support this effort under Task 114 of the CORA Standing Offer.

The objectives of project were to analyze the current Tyche system, identify the problems affecting simulation execution time, and make recommendations to speed up the Tyche simulation execution.

Results: The results of Tyche system analysis consists of a conceptual model, a dynamic component model and detailed function flowcharts of simulation process. The logic model of Tyche was extracted and represented as entities, concepts, dynamic components and their organizations. The Tyche simulation process comprised three steps, i.e. pre-iterations, iterations and post-iterations.

Based on the Tyche system analysis, two kinds of problems were identified, including the problem of single thread execution and the issues of repeated operations in iterations.

Significance: A parallel architecture was proposed to reduce simulation execution time. To maximize the compatibility with current Tyche system, this architecture kept the existing Tyche Editor & Dashboard for user iteration and simulation monitoring. To speed up simulation execution, a new module called Tyche Parallelization Management Tool was defined and integrated into the architecture, which was used for partitioning, mapping, generating and managing parallel iterations of simulation instances. An optimized simulation engine was also suggested to speed up simulation execution further.

Information was collected and evaluated for 350 discrete event simulation tools/libraries/languages based on defined criteria. Finally, two groups of alternative programming environments for future Tyche development were recommended including (1) MS MPI with visual studio & Simio, and (2) AnyLogic, Simul8, SimEvents and SimPy.

Additional tasks for a further evaluation of the functionality and literature-based speed comparison among the simulation development environments were also completed by the project team. The results showed that VB, VC++ and VC# can provided more functions in the functionality checklist provided by the Scientific Authority than the integrated high-level development environments. For the speed comparison, no test case was found for the integrated high-level development environments, including AnyLogic, SimEvents, Simio and Simul8. 21 references of speed comparison related to the low-level programming languages were collected, analyzed and summarized. The results of speed comparison showed that VC++ is the fastest programming language. VB and VC# are quite similar, Java is a little bit slower than VB & VC#, and Python is the slowest language because it is an interpretation-based language.

Future plans: Based on the Tyche analysis and recommendations, the following tasks are identified as next steps, including implementing the proposed parallel architecture, developing an optimal simulation engine, and testing as well as evaluating the top programming environments in detail.

Sommaire

Analysis of the Tyche Simulation Engine and Recommendations for Future Development

R. J. Guo; J. Brooks; DRDC CORA CR 2012-081; R & D pour la défense Canada – CORA; Avril 2012.

Introduction ou contexte: Le système de simulation Tyche est fondé sur une simulation de Monte-Carlo d'événements discrets développé par le Centre d'analyse et de recherche opérationnelle (CARO) de Recherche et développement pour la défense Canada (RDDC) pour effectuer l'analyse structurelle des forces dans le cadre des flottes pour appuyer la planification stratégique.

En utilisant le système logiciel Tyche actuel, une simulation typique des forces prend plusieurs heures sur une plateforme avec un seul processeur. Une priorité actuelle de maintenance est de réduire le temps d'exécution des simulations pour permettre de vérifier le plus rapidement possible un grand nombre d'options de composition de la flotte. En particulier, nous devons refondre l'architecture pour prendre en charge le traitement en parallèle dans un environnement distribué en grappe ou en grille, puisque le CARO va acquérir un environnement informatique haute performance. Un contrat a été attribué à CAE Professional Services (Canada) Inc. pour appuyer ces travaux dans le cadre de la tâche 114 de l'offre à commande du CARO.

Le projet avait pour but d'analyser le système de simulation Tyche actuel, de cerner les problèmes qui nuisent à la vitesse d'exécution et de formuler des recommandations en vue d'accélérer l'exécution de simulations Tyche.

Résultats: L'analyse du système Tyche a produit un modèle conceptuel, un modèle dynamique des composants et des organigrammes détaillés des fonctions du processus de simulation. Le modèle logique qui sous-tend Tyche a été extrait et représenté sous forme d'entités, de concepts, de composants et de leur organisation. Le processus de simulation Tyche comprend trois étapes : traitement avant les itérations, itérations, traitement après les itérations.

Les résultats de l'analyse du système Tyche ont permis de cerner deux types de problèmes : utilisation d'un seul fil d'exécution et opérations répétées au cours des itérations de simulation.

Importance: Nous proposons l'utilisation d'une architecture parallèle pour réduire le temps d'exécution des simulations. Pour assurer la compatibilité avec le système Tyche actuel, dans la mesure du possible, la nouvelle architecture garde l'éditeur et le tableau de bord actuels de Tyche pour la surveillance par l'utilisateur des itérations et des simulations. Pour accélérer les simulations, un nouveau module, le Tyche Parallelization Management Tool (outil de gestion de la parallélisation de Tyche), a été défini et intégré à l'architecture. Cet outil permet la division, le mappage, la génération et la gestion d'itérations parallèles d'instances de simulation. Nous proposons aussi d'optimiser le moteur de simulation pour accélérer encore plus les simulations.

Nous avons recueilli et évalué, en fonction de critères définis, des renseignements sur 350 outils, bibliothèques et langues de simulation d'événements discrets. Finalement, nous recommandons

deux groupes de nouveaux environnements de programmation pour le développement futur de Tyche : d'une part, MS MPI avec Visual Studio et Simio et, d'autre part, AnyLogic, Simul8, SimEvents et SimPy.

L'équipe de projet a aussi mené à bien des tâches supplémentaires d'évaluation de la fonctionnalité et, à partir de la littérature, de comparaison de la vitesse dans des environnements de développement de simulations. Les résultats obtenus indiquent que VB, VC++ et VC# peuvent fournir un plus grand nombre de fonctions de la liste de contrôle établie par l'autorité scientifique que les environnements de développement intégrés de haut niveau. Quant à la comparaison de la vitesse, aucun scénario d'essai n'a été trouvé pour les environnements de développement intégrés de haut niveau (AnyLogic, SimEvents, Simio, Simul8, etc.). Nous avons recueilli, analyser et résumé 21 références relatives à la comparaison de la vitesse pour les langues de programmation de bas niveau. Les comparaisons de la vitesse démontrent que VC++ est la langue de programmation la plus rapide. VB et VC# ont une vitesse semblable, Java est un peu plus lent que ceux-ci et Python est la plus lente, puisque c'est une langue interprétée.

Perspectives: En vertu de l'analyse de Tyche et des recommandations formulées, nous avons identifié les tâches suivantes comme prochaines étapes : mise en œuvre de l'architecture parallèle proposée, développement d'un moteur de simulation optimisé, mise à l'essai et évaluation détaillées des principaux environnements de programmation.

Table of contents

Abstract …….. i
Résumé …..... i
Executive summary iii
Sommaire v
Table of contents vii
List of figures ix
List of tables x
1 Introduction..... 1
1.1 Background..... 1
1.2 Objectives and scope 1
1.3 Constraints..... 1
1.4 Document organization 2
2 Methodology..... 3
3 Results of Tyche System Anaysis..... 5
3.1 Tyche Logic Model and Simulation Process..... 5
3.1.1 Tyche Dynamic Components and Organization 5
3.1.2 Overview of Simulation Process..... 6
3.1.3 Pre-iterations 7
3.1.4 Iterations..... 8
3.1.4.1 ResetAssetData Function in Iterations..... 9
3.1.4.2 GenerateEvents Function in Iterations..... 9
3.1.4.3 ScheduleAssets Function in Iterations..... 10
3.1.4.4 OutputData Function in Iterations 12
3.1.5 *Post-iterations* – Statistical Data Analysis..... 13
3.2 Problems Affecting Simulation Execution Time..... 14
3.2.1 Problem of Single Thread Execution 14
3.2.2 Issues within Iterations..... 15
3.3 Summary of Tyche System Analysis..... 16
4 Recommandations..... 17
4.1 Introduction 17
4.2 Parallel architecture 17
4.2.1 Overview of Parallel and Distributed Discrete Event Simulations 17
4.2.2 Proposed System Architecture..... 18
4.2.2.1 Parallel Architecture 18
4.2.2.2 Detailed Architecture with Simple Partitioning 19
4.2.2.3 Detailed Architecture with Balanced Partitioning..... 21

4.2.2.4 Architecture with Two Levels of Parallel Execution 22
4.2.3 Partitioning and Mapping of Iterations in Simulation Instances 23
4.2.3.1 Overview of Parallel Programming 23
4.2.3.2 Algorithms for Partitioning Iterations in Simulation Instances .. 24
4.2.3.3 Mapping Tyche Modules onto Jobs, Tasks and Computer Nodes on Windows HPC Server 2008 28
4.3 Optimizing the Design of the Simulation Engine 31
4.3.1 Optimizing *Iterations* to Speed up Simulation 31
4.3.1.1 Optimizing the *GenerateEvents* Function in Iteration 31
4.3.1.2 Optimizing the *ScheduleAssets* Function in Iteration 32
4.3.1.3 Optimizing the *OutputData* Function in Iteration 34
4.3.1.4 Summary of Optimized *Iterations* in the Simulation Engine 35
4.3.2 Using Parallel Functions to Speed up Simulation 35
4.3.2.1 Parallelizing Functions used in *Pre-iterations* 36
4.3.2.2 Parallelizing Functions used in *Iterations* 37
4.3.2.3 Parallelizing Functions used in *Post-iterations* 38
4.4 Evaluating and Recommending Programming Environments 38
4.4.1 Establishing Criteria for Evaluating Programming Environments 38
4.4.2 Gathering information about programming environments for DES/PDES ... 38
4.4.3 Evaluating Programming Environments 39
4.4.4 Recommending Programming Environments 40
5 Further evaluation of functionality and speed of simulation development environments 42
5.1 Functionality review of simulation development environments 42
5.2 Speed comparison of simulation development environments 45
6 Summary and Next Steps 57
References 59
Appendix A: List of Libraries/Software/Tools/Lanauges for DES and PDES 69
Appendix B: Top 10 libraries/software/tools/lanauges for DES and PDES 72
Appendix C: Results of functionality reveiw of simulation development environments 77
List of Symbols/Abbreviations/Acronyms/Initialisms 111

List of figures

Figure 1: Organization of dynamic components in Tyche 6
Figure 2: Flowchart of the Tyche simulation engine 7
Figure 3: Pre-iterations 8
Figure 4: Overview of iteration process 8
Figure 5: ResetAssetData function called within iteration 9
Figure 6: GenerateEvents function called within iteration 10
Figure 7: ScheduleAssets function used in iteration 11
Figure 8: OutputData function used in iteration 13
Figure 9: Statistical data analysis 14
Figure 10: Single-thread structure in Tyche 15
Figure 11: Proposed architecture for speeding up the Tyche simulation execution 19
Figure 12: Detailed architecture with simple partitioning 20
Figure 13: Detailed architecture with balanced partitioning 21
Figure 14: Architecture with two-level parallel execution 22
Figure 15: Sample algorithm for approximately-balanced partitioning 25
Figure 16: Partitioning and mapping of simulations and iterations 26
Figure 17: Sample algorithm for balanced partitioning 27
Figure 18: Optimizing GenerateEvents function in iteration 32
Figure 19: Optimizing ScheduleAssets function in iteration 33
Figure 20: Optimizing OutputData function in iteration 34
Figure 21: Optimized DoIteration in the simulation engine 35
Figure 22: Using parallel functions to speed up simulation 36
Figure 23: Group A; Average execution time (in milliseconds) of the test cases without arithmetic operations 49
Figure 24: Group B: Average execution time of the test cases for arithmetic and trigonometric operations. 50
Figure 25: Results of speed comparison between the six related programming languages 55
Figure 26: Results of speed comparison between VB, VC++ and VC# 56

List of tables

Table 1: Single-level parallel mapping 28
Table 2: Two-level parallel mapping 30
Table 3: Top 42 + 1 tools with the most citations 39
Table 4: Results of functionality comparison between the Tyche checklist and the related simulation development environments 43
Table 5: Average execution time, relative execution time and relative speed factors of programming languages in the Bruckschlegel experiment 51
Table 6: Summary of the relative speed factors of the related programming languages 51
Table 7: Relative speed factors compared to that of Visual Basic 53
Table 8: Computation process of overall speed of Visual Basic 53
Table 9: Speed and overall speed of programming languages 54
Table 10: Overall speed of the six related programming languages 54
Table 11: Result speed for the VB, VC++ and VC# group 55

1 Introduction

1.1 Background

The Tyche simulation system is a Monte Carlo based discrete event simulation system developed by Defence Research and Development Canada's (DRDC) Center for Operational Research and Analysis (CORA) for conducting force structure analysis at the fleet level in support of strategic planning (Heppenstall, 2007; Allen, et al., 2006; Michalowski, 2009).

With the Tyche software system, a typical force structure simulation may take several hours to complete on a single-processor platform. A current priority is to shorten the simulation runtime to increase the capacity to test a large number of fleet options in the minimal amount of time. Specifically, redesigning the architecture to support parallel processing in a cluster/grid environment is sought as CORA has acquired a high-performance computing environment. CAE Professional Services (Canada) Inc. (CAE PS) was contracted to support this effort under Task 114 of the CORA Standing Offer.

1.2 Objectives and scope

The goal of this project is to analyze the Tyche simulation system and recommend new system architecture and approaches to speed up Tyche simulation execution. The project objectives are as follows:

- Analyze the current Tyche simulation system,
- Identify the problems that affect simulation execution time,
- Propose architecture, approaches and techniques to speed up simulation execution, and
- Evaluate and recommend programming environments for future Tyche development.

The scope of this project considers the four aspects listed above. Namely, based on the analysis of current Tyche system, this report provides a summarized logic model and simulation process of Tyche system as well as identified problems. To solve the problems found, a new architecture, an optimized simulation engine and a group of alternative programming environments will be recommended to reduce simulation execution time and support future Tyche development.

1.3 Constraints

Tyche is a software system that has evolved for seven years, with 561 files and is 42.9 MB in size. Existing Tyche documents mainly focus on how to use the Tyche system, rather than the conceptual model and details of software design. This project is focused on the recommended architecture. The approaches and techniques mainly emphasize the ideas to solve problems and speed up simulation execution, rather than the details of software implementation. Detailed software design and implementation may be pursued in future projects.

Regarding the evaluation and recommendation of programming environments, the focus was on feature extraction. The sorting and comparison of tools are based on the documents/manuals/release notes or review papers from related product web sites or other Internet sources. Detailed function/speed testing and comparison of the recommended programming environments could be completed in a separate project.

1.4 Document organization

This report is organized as follows.

Section $_{two}$ briefly describes the methodology employed in this project.

Section $_{three}$ deals with the results of Tyche system analysis, which consist of the extracted Tyche dynamic component model, the detailed simulation process and the identified problems affecting simulation execution time.

Section $_{four}$ depicts the recommendations for speeding up Tyche simulation execution, which comprise a parallel architecture, a group of approaches for optimizing the design of the simulation engine and a set of alternative programming environments for future Tyche development.

Section $_{five}$ summarizes this project and identifies next steps.

2 Methodology

This section provides a very brief summary of the methods that were used and the activities conducted to complete this project.

The following methods were used to conduct this project:

- System and software analysis,
- Functional identification and extraction,
- Summarization,
- Problem identification,
- Scoping, objectives and requirements development,
- Task analysis/decomposing,
- Software, algorithm and optimization design,
- Literature collection, identification and review,
- Evaluation criteria development,
- Feature recognition, extraction and comparison, and
- Quality assessment, sorting and selection.

This project identifies the following sub-tasks to achieve its objectives:

- Extracting a Tyche logic model and analyzing the Tyche simulation process,
- Identifying problems affecting simulation execution time,
- Proposing a parallel architecture to reduce simulation execution time,
- Developing approaches to optimize the design of the simulation engine and further speed up simulation execution, and
- Evaluating and recommending programming environments for future Tyche software development.

The analysis of Tyche system was performed by reading the Tyche user guides, developing and executing application examples with the Tyche software, analyzing input & output data, and extracting static and dynamic logic models of current the Tyche system. The analysis of the Tyche simulation process was completed by reading source code, extracting main steps, and forming detailed function flowcharts.

The problems affecting simulation execution time were identified by summarizing the results of the Tyche system analysis and software execution with examples, and combined with the detailed studies of simulation engine source code.

The proposed architecture is formed by combining various methods, including scoping & requirement development, task & functional analysis, software design, parallel programming and algorithm design. To reduce simulation execution time further, this project optimizes the design of the simulation engine by identifying, extracting and reducing redundant or repeated operations in iterations, as well as using parallel programming techniques.

To evaluate and recommend programming environments, this project team in collaboration with DRDC CORA, first defined a group of criteria for quality assessment. The project team then collected the surveys, reviews and summaries related to the libraries, languages, frameworks, Application Programming Interfaces (APIs) and software/tools for discrete event simulation from the Internet. Based on the popularity of citations in surveys/reviews/summaries, top 43 tools were identified for further evaluation. With the extraction and comparison of detailed features of the top 43 tools, the final recommendations of programming environments were formed.

3 Results of Tyche System Anaysis

This section provides a description of the Tyche system including its logic model and simulation process, for identifying potential problems and developing solutions. The section first introduces the extracted dynamic component model and detailed function flowcharts of the simulation process in the Tyche system, and then identifies the problems that affect the Tyche simulation execution time.

3.1 Tyche Logic Model and Simulation Process

The analysis process of the Tyche logic model consists of extracting physical, conceptual and dynamic entities/components, and identifying the relationships among various entities and components. The analysis of the Tyche simulation process is completed by reading source code, extracting main steps and forming detailed function flowcharts.

3.1.1 Tyche Dynamic Components and Organization

The dynamic components in the Tyche system consist of simulation instances, iterations, scenarios, phases, finite state machines of asset levels, and events. Figure 1 shows the organization of the Tyche dynamic procedural components. A run of the Tyche software system might include a group of simulation instances for various force structure models, where a simulation instance comprises thousands of iterations, each instance being an execution of the Tyche Simulation Engine.

An iteration includes a group of mission-related scenarios, such as, peace keeping, and a set of non-mission-related asset tasks, such as maintenance. A scenario consists of a sequence of phases, each of which performs a sub-task in mission. A phase contains a set of demand capabilities provided by assets, and each asset links to a finite-state machine consisting of various levels of employment. An asset level may demand other assets for certain capabilities.

The arrows linking to scenario phases, assets and asset levels in Figure 1 represent events. An event is a time-associated occurrence for activating a scenario phase or an asset level, or implementing a transition from a scenario phase to another phase, or from an asset level to another level. It is the events that advance time and drive the simulation process in each iteration.

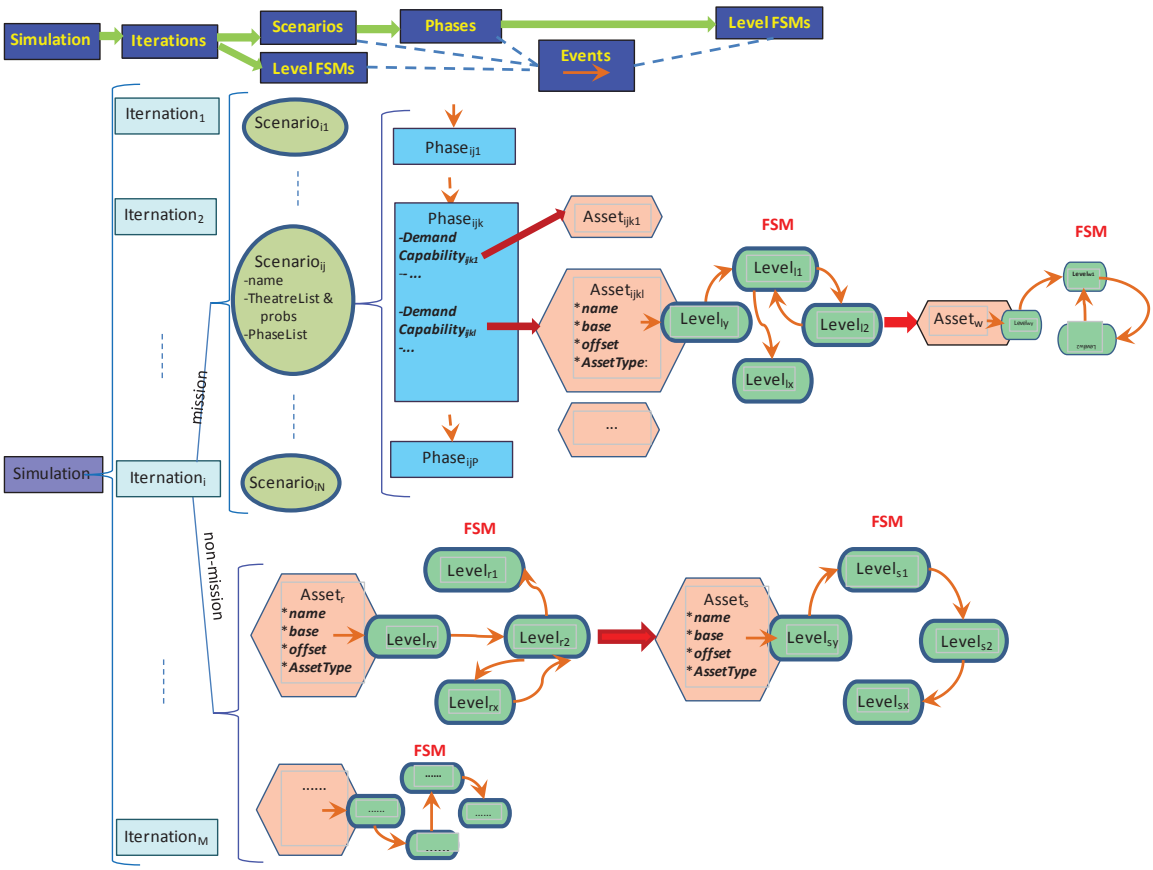

Figure 1: Organization of dynamic components in Tyche

3.1.2 Overview of Simulation Process

The overview of the Tyche simulation process is shown in Figure two (Tyche Help Files, 2011), which consists of three steps: *Pre-iterations*, *Iterations* and *Post-iterations*. The *Pre-iterations* are used to initialize the system, load a user force structure model and convert the force structure model into internal data structures. The *Iterations* are the core procedure in the Tyche simulation system, which generate dynamic events and drive simulation execution. The *Post-iterations* are responsible for statistical data analysis of simulation results.

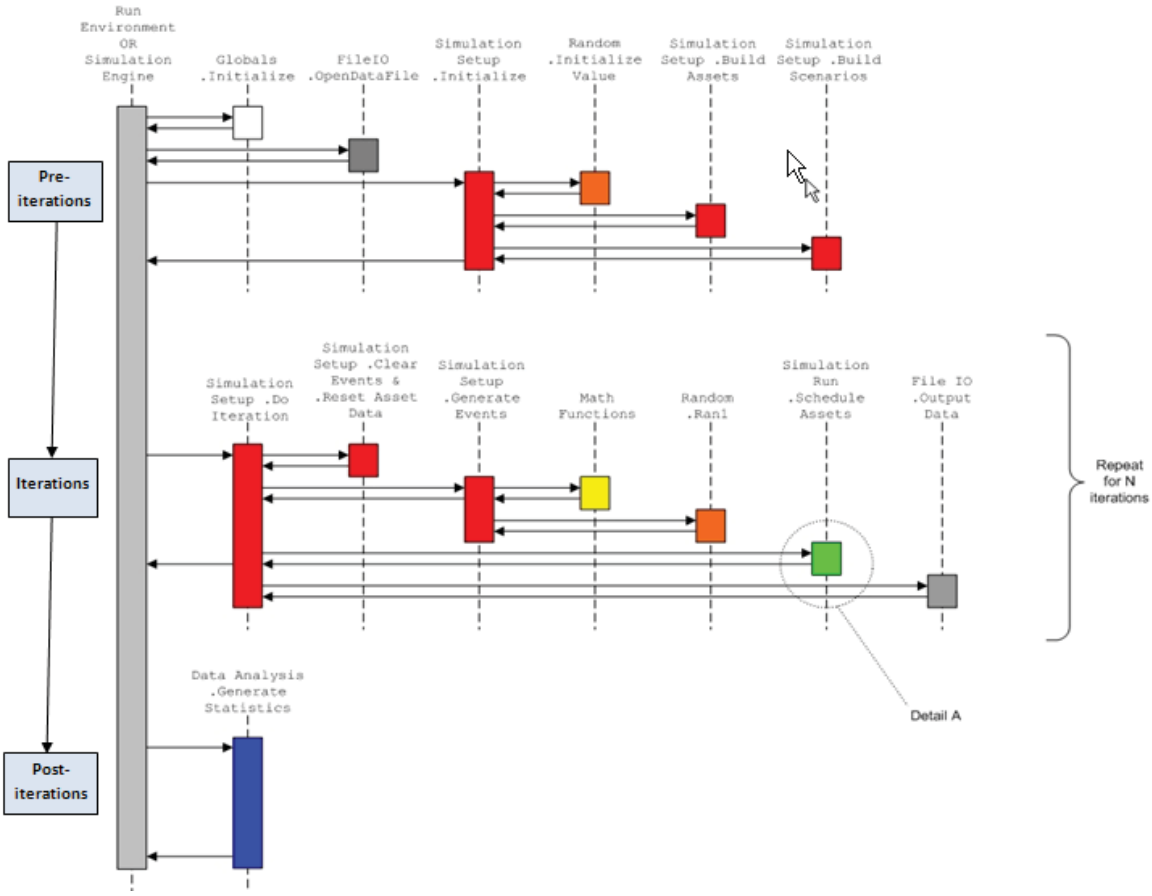

Figure 2: Flowchart of the Tyche simulation engine

3.1.3 Pre-iterations

Figure three shows the extracted function flowchart of *Pre-iterations*. At first, the system initializes global variables, and loads the specified user force structure model. It then creates a random seed for the simulation execution. Finally, it generates various data structures for assets and scenarios used in simulation process.

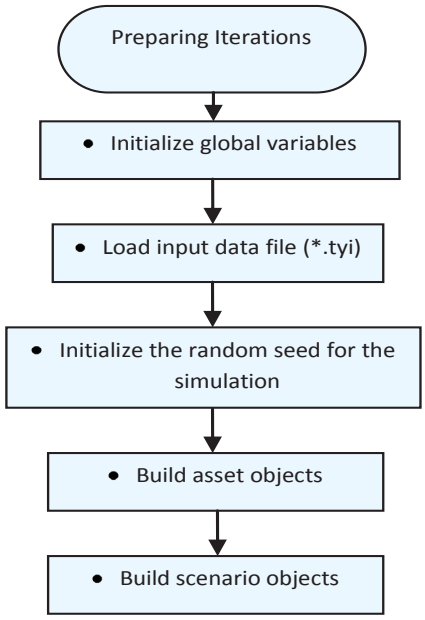

Figure 3: Pre-iterations

3.1.4 Iterations

Iterations are a significant step in the Tyche simulation process. An iteration is an execution of a force structure with the years to simulate and a random start date. Figure four shows the functions or sub-procedures used in *Iterations*, including *ResetAssetData*, *GenerateEvents*, *ScheduleAssets*, and *OutputData* functions. The *ResetAssetData* function is used to initialize an iteration. The *GenerateEvents* function creates all events associated with all the scenario phases and asset levels with "Schedule" or "Random" types. The function of *ScheduleAssets* searches demanded and available assets for all events and advances the simulation date to handle all events. The last function, *OutputData*, is employed to form output messages for the current iteration and associated events, and write them to an output file (*.tyo).

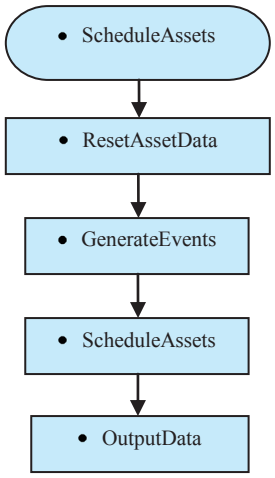

Figure 4: Overview of iteration process

3.1.4.1 ResetAssetData Function in Iterations

Figure five is the function flowchart of *ResetAssetData* function. Obviously, the main work in this function is to clean the history and future event lists, and create initial events for default levels of all internal and external assets.

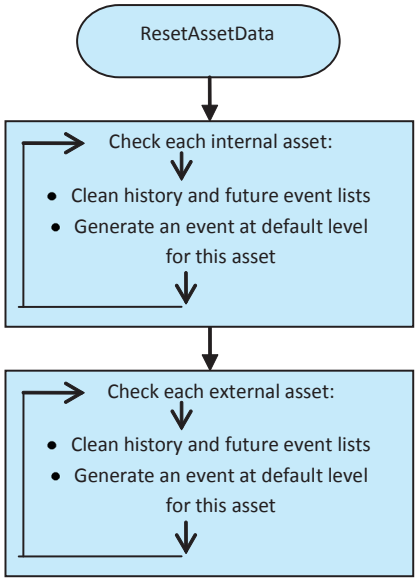

Figure 5: ResetAssetData function called within iteration

3.1.4.2 GenerateEvents Function in Iterations

The *GenerateEvents* function used in *Iterations* is to generate events for all the scenario phases and assets levels with "Schedule" or "Random" types, as shown in Figure 6.

After declaring basic variables, the *GenerateEvents* function creates a random *IterationBeginDate* for the current iteration.

Each iteration may consist of two groups of tasks: one is the scenario-related mission and another is the non-mission-related tasks at asset levels. This function goes through each level of each asset in the system to see if the level is a "Schedule" or "Random" type. For all the asset levels with "Schedule" types, the function, first, computes the times of occurrences with frequency and years to simulate, and the *StartDates* of the level based on the *IterationBeginDate*, offset, duration, and frequency. It then calculates the durations of levels based on a triangular distribution. It next generates all possible events for this asset level and uses the first event's *StartDate* and the duration of each successive event to compute the *StartDates* of all the events for the asset level. If the asset level is a "Random" type, the function generates random events and *StartDates* for the level.

This function uses the similar way to create all possible events for each scenario phase. If a scenario phase is a "Schedule" or "Random" type, it computes the times of occurrences with frequency and years to simulate, generates an event for each occurrence, and creates *StartDates* for all events based on the *IterationBeginDate* and phase duration. The function calculates durations based on a triangular or uniform distribution, and adjusts durations with the overlap specified by users. This function also recursively checks the "Following Phase" of the phase, and the "Following Phase" of "Following Phase" to generate all possible events for successive phases. If the phase is a "Random" type, then it generates a random *StartDate* for this phase, computes

the number of occurrences with the frequency and simulated years, creates an event for each possible occurrence, and uses the first event's *StartDate* and the duration of each successive event to compute the *StartDates* of all events for the phase.

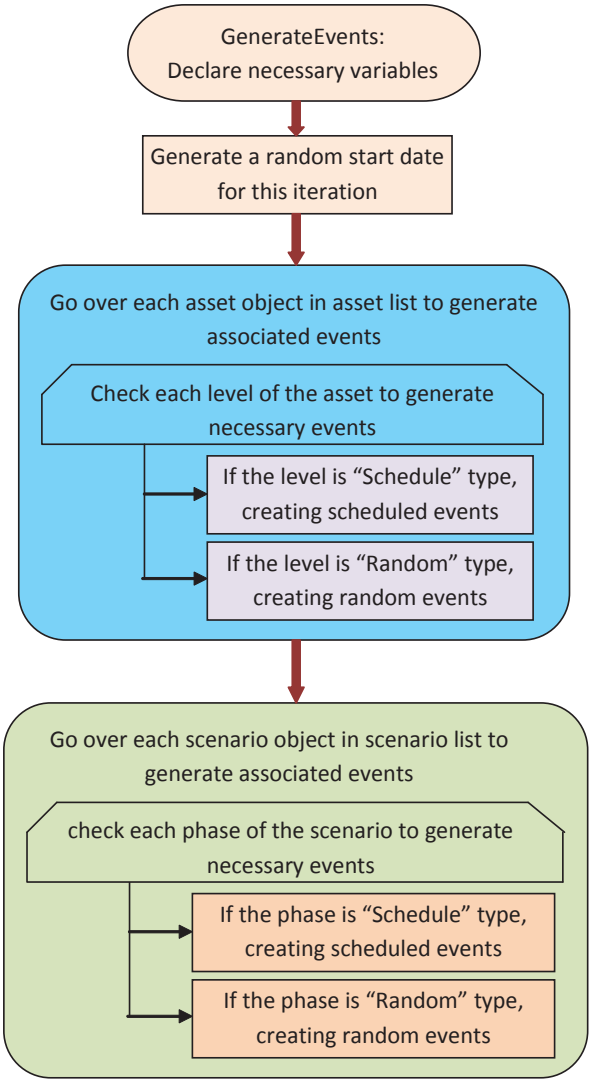

Figure 6: GenerateEvents function called within iteration

3.1.4.3 ScheduleAssets Function in Iterations

The *ScheduleAssets* function is used to handle all events in an iteration. The extracted function flowchart is shown in Figure 7. For each event in iteration, this function identifies and selects demanded and available assets, and then links the selected assets to the event.

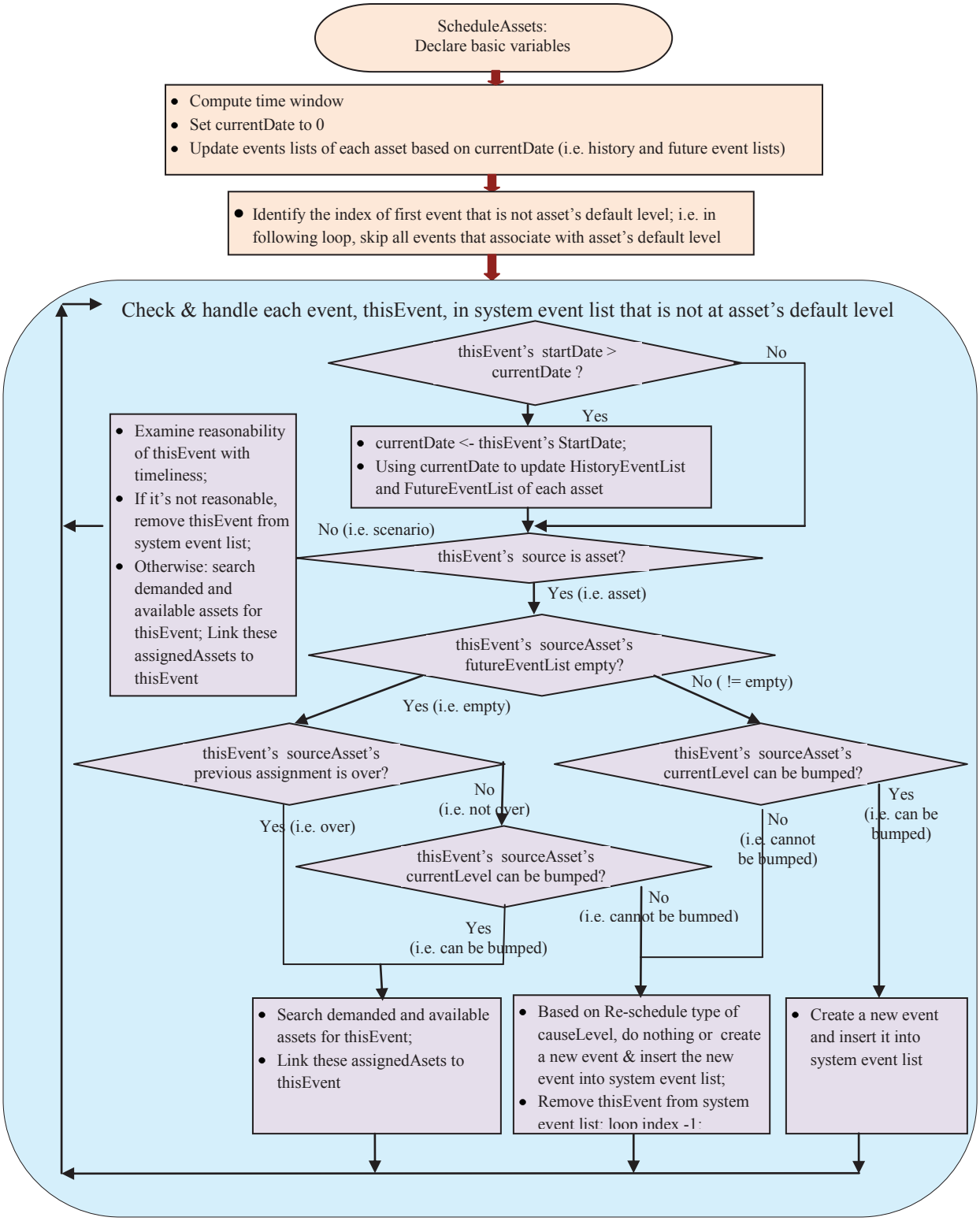

Figure 7: ScheduleAssets function used in iteration

This function, first, computes the time window based on the simulated years and sets the *currentDate* to zero. It then updates the *HistoryEventList* and *FutureEventList* of each asset with *currentDate* and moves all the events with *StartDates* equal to or less than the *currentDate* from its *FutureEventList* to *HistoryEventList*.

Next, this function checks each event, named *currentEvent*, in the system event list to see if its' *StartDate* is greater than the *currentDate*. If it is yes, the system advances the *currentDate* with this event's *StartDate*.

If the source of *currentEvent* is a scenario, then this function reviews the reasonability of *currentEvent* with timeliness of the cause phase. If it is not reasonable, such as its end date is outside the time window of iteration, then the function ignores the *currentEvent*, removes it from the system event list and continues to check next event. Otherwise, it searches demanded and available assets for *currentEvent*, and links the assigned assets to *currentEvent*.

If the source of *currentEvent* is an asset, then the function checks the asset's *FutureEventList*. If the *FutureEventList* is empty, then it checks if this asset's previous assignment is over. If it is over, then the function searches demanded and available assets for *currentEvent* and links the assigned assets to *currentEvent*, and computes or adjusts various dates associated with these assets. If the previous assignment is not over, then the function checks if the *currentLevel* of the source asset can be bumped. If it can be bumped, then the function searches demanded and available assets for *currentEvent*, and links the assigned assets to *currentEvent*. If it cannot be bumped, then the function checks the re-schedule type of *currentLevel* of the source asset to determine if a new event should be generated for re-scheduling the *currentLevel*.

If the *FutureEventList* of the source asset is not empty, this function checks if the *currentLevel* of the source asset can be bumped. If it can be bumped, then, the function creates a new event and inserts it into system event list, and continues to handle next event in the system event list. If it cannot be bumped, then the function checks the re-schedule type of *currentLevel* of the source asset, to determine if a new event should be generated for re-scheduling of the *currentLevel*.

3.1.4.4 OutputData Function in Iterations

The *OutputData* function in *Iterations* is used to form output data and write the data to output files. Figure $_{\text{eight}}$ shows the details of *OutputData* function. If the current iteration is the first iteration in a simulation instance, this function opens a text file stream and writes the information of simulation instance, such as the number of iterations, simulated years, information of assets, levels, scenarios and phases, as well as asset types.

For each iteration, this function extracts various attributes of each event and associated assets, forms a message buffer for the current event and writes the message buffer to output file.

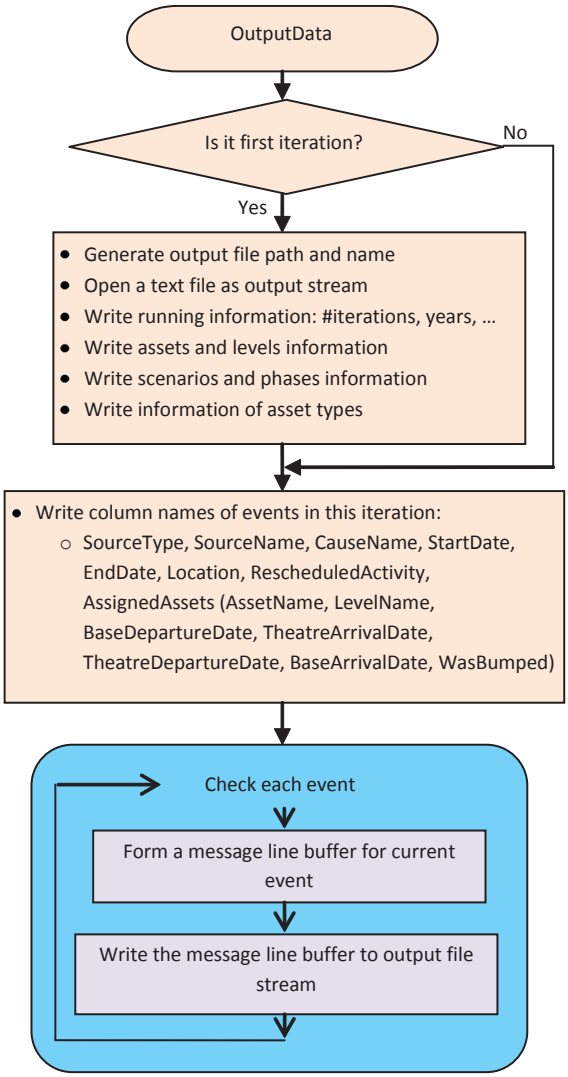

Figure 8: OutputData function used in iteration

3.1.5 *Post-iterations* – Statistical Data Analysis

Figure nine shows the flowchart of statistical data analysis in *Post-iterations*. This function, first, opens an output file stream for the results of data analysis, and it, then, initializes datasets of assets, phases and risk. After collecting data for a simulation output file, the function generates four categories of statistical data for users: asset statistics, scenario statistics, capability statistics and risk.

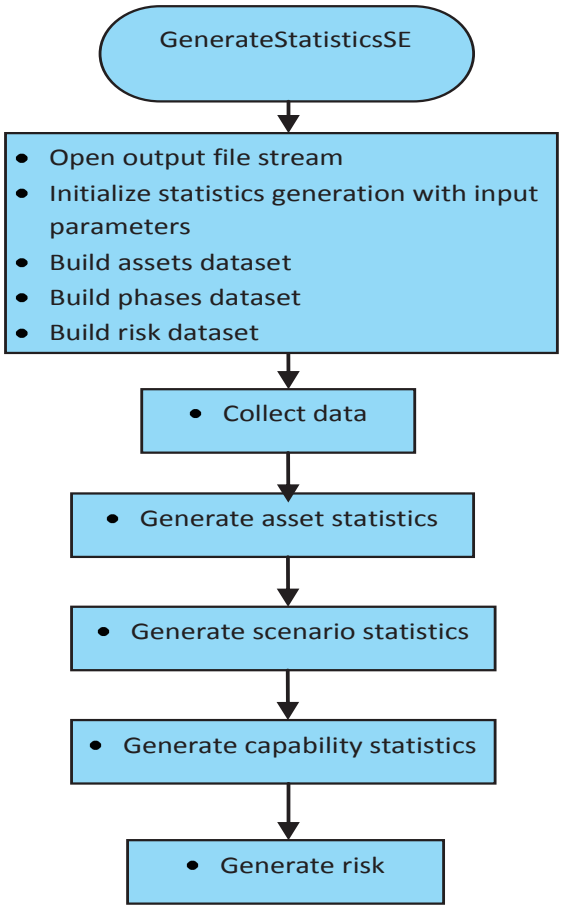

Figure 9: Statistical data analysis

3.2 Problems Affecting Simulation Execution Time

Based on the analysis of the Tyche simulation system, two categories of problems were identified, including the problem of single thread execution and issues within the iterations.

3.2.1 Problem of Single Thread Execution

Figure 10 shows the diagram of the Tyche modules. The "Tyche-GUI" is a Graphic User Interface (GUI) for developing and testing force structure models, and the Dashboard is a simulation management tool used to generate, execute and monitor multiple simulation instances, each of which consists of a large number of iterations. The current Tyche software system is single-thread-based. This means that all iterations in simulation instances are executed one by one on a single thread on a single computer processor. The single-thread execution results in significant time consumption. For example, Tyche spent eight hours to complete a typical simulation example used in FMS II, with 1000 iterations and seven simulated years on a typical desktop.

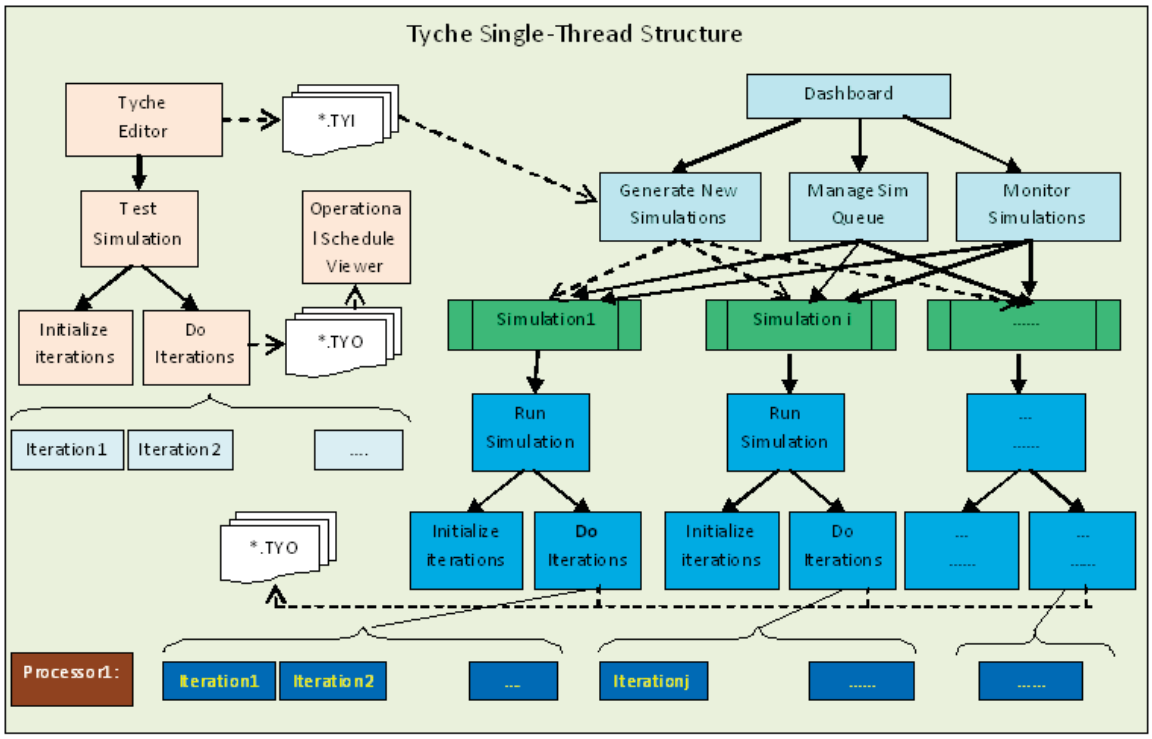

Figure 10: Single-thread structure in Tyche

3.2.2 Issues within Iterations

This project also analyzes problems within iterations. As mentioned in previous sections, there are four main functions used in a single iteration process, including *ResetAssetData*, *GenerateEvents*, *ScheduleAssets* and *OutputData* functions. Based on the detailed analysis of source code in the functions, some redundant/repeated operations or frequent file access are found, which also affects execution time.

1. **Initial events for default asset levels:** In Tyche, an event is defined as a time-related occurrence that either activates a scenario/asset level, or implements a transition from a scenario phase to another phase or from an asset level to another level. The *ResetAssetData* function creates an initial event for each asset at its default level. The default "idle" state of an asset would not need an initial event to activate at the beginning of simulation, although the OpSched Viewer and asset statistics do require these initial events.

2. **Repeated operations in *GenerateEvents* function:** In the function of *GenerateEvents*, there are repeated operations in all iterations, including going through each level of each asset in the system asset list, and each phase of each scenario in the system scenario list to generate initial events. The basic fact is that, for all iterations, the system uses the same force structure model. It means that the relationships between scenario phases, and between asset levels do not change from iteration to iteration for a given force structure model. The repeated operations for going through the phase and asset levels in each iteration to find such structure relationships for possible events is time consuming and not necessary, because a basic event list for a given force structure model could be shared by all iterations in a given simulation instance, like the asset list shared by all iterations in a simulation instance in the current Tyche system.

3. **Repeated operations in *ScheduleAssets* function:** For all iterations in a simulation instance, the function of *ScheduleAssets* searches demanded assets again and again from iteration to iteration for the same capability demand relationships in the same force structure model. Such searching is also very time-consuming and not necessary, because a group of candidate assets for the demand capabilities in each basic event and iteration could also be shared by all the iterations in a given simulation instance.

4. **Frequent file output operations in *OutputData* function:** The current *OutputData* function forms and writes information to output file once for each event. Such frequent file output operations are also time-consuming.

3.3 Summary of Tyche System Analysis

In summary, this project analyzed and extracted the basic conceptual model, dynamic components and relationships, as well as the detailed function flowcharts in the simulation process in the Tyche system.

Based on the Tyche system analysis, two kinds of problems that affect simulation execution time were identified, including the problem of single-thread execution, and the issues of redundant/repeated operations in iterations. A group of solutions are proposed in the next Section to solve these problems.

4 Recommandations

This section describes the recommendations made for the future Tyche simulation system.

4.1 Introduction

As expressed by the Technical Authority, the goals of the project were to recommend a new system architecture to speed up the Tyche simulation execution and to propose a list of candidate programming environments for its implementation.

Specifically, the tasks provided by Technical Authority's were to:

- Identify a new system architecture that:
    - Supports all the functionality in the current Tyche simulation system, including tools for developing and testing force structure models, conducting simulation execution with a great number of iterations, and creating, managing, and monitoring multiple simulation instances;
    - Is compatible with the current Tyche system and Windows High-Performance Computing (HPC) Server 2008 R2;
    - Implements parallel technology to increase the speed of code execution; and
    - Minimizes the simulation execution time by implementing parallel technology and by optimizing the iteration algorithms in the simulation engine; and
- Identify and evaluate appropriate programming environments for the implementation of this new architecture.

Three categories of recommendations are proposed to reach the objectives, including a parallel architecture, a group of approaches for optimizing the design of the simulation engine and a set of alternative programming environments for Tyche software development. Each is discussed in the following sections.

4.2 Parallel architecture

4.2.1 Overview of Parallel and Distributed Discrete Event Simulations

A simulation is a system that represents or emulates the behaviour of another system over time (Fujimoto, 2000; Banks, et al., 2009). In a computer simulation, the system doing the emulating is a computer program, and the system being emulated is called the physical system. The physical system may be an actual, realized system, or it may only be a hypothetical one, for example, one of several possible design alternatives that only exist in the mind of its inventor.

Discrete event simulation (DES) is characterized by discrete-state models (as opposed to continuous-state models) and the event-driven approach (Banks, et al., 2009; Schriber & Brunner, 2010; Ferscha, 1995; Fujimoto, 2000; Vee & Hsu, 1990; Law & Kelton, 1991; Jain, 1991; Ören, 2002).

Parallel discrete-event simulation (PDES) refers to executing a single discrete-event simulation program on parallel computers, which can be shared-memory multiprocessors, distributed-memory clusters of interconnected computers, or a combination of both (Fujimoto, 2000;

Tropper, 2002; Dedenhoeffer, et al., 2002; Liu, 2009, Perumalla, 2006; Dave, 2005; Sulistio, et al., 2002). By exploiting the potential parallelism in simulation models, PDES can overcome the limitations imposed by sequential simulation in both execution time and the memory space. As such, PEDS can bring substantial benefits to time-critical simulations, as well as simulations of large-scale systems that demand an immense amount of computing resources.

A parallel or a distributed simulation is typically composed of a collection of sequential simulations, each modelling a different part of the physical system and (at least potentially) executing on a different processor, such as replicated trails (Henderson, 2003; Hybinette & Fujimoto, 2001), functional decomposition (Foster, 1995; Comfort, 1984;), space-time view (Chandy & Sherman, 1989; Bagrodia, et al., 1991), time-parallel (Heidelberger & Stone, 1990; Nicol, et al., 1994; Ammar & Deng, 1992; Lin & Lazowska, 1991; Kiesling & Luthi, 2005), and space-parallel (Liu, 2010; Lamport, 1978). Many details of parallel and distributed simulation and synchronization algorithms can be found in the literatures (Fujimoto, 2000, 1990; Lin & Fishwick, 1996; Low, et al., 1999; Nicol & Fujimoto, 1994; Nicol & Liu, 2002; Wang, et al., 2009; Prassad & Deo, 1991; Taylor, et al., 1993; Malik, et al., 2010; Page Jr, 1994).

4.2.2 Proposed System Architecture

4.2.2.1 Parallel Architecture

To speed up the Tyche simulation execution, a parallel system architecture is proposed, as shown in Figure 11. There are four modules at the highest level in the architecture, including the Tyche Model Development Tool (TMD), the Tyche Simulation Engine (TSE), the Tyche Simulation Management Tool (TSM), and the Tyche Parallelization Management Tool (TPM).

The Tyche Model Development Tool is used to create, edit and test user force structure models. The Tyche Simulation Engine is a kernel module for simulation execution. The Tyche Simulation Management Tool is mainly employed for creating, managing and monitoring multiple simulation instances. Finally, the Tyche Parallelization Management Tool is responsible for partitioning iterations of simulation instances into iteration groups, mapping the groups onto cluster computer nodes and processors/cores, managing parallel execution of iteration groups, and generating final result output files. In this architecture, the Windows High-Performance Computing (HPC) Server 2008 R2 is used as a parallel platform (Microsoft, 2010a, 2010b, 2011, 2008a, 2008b).

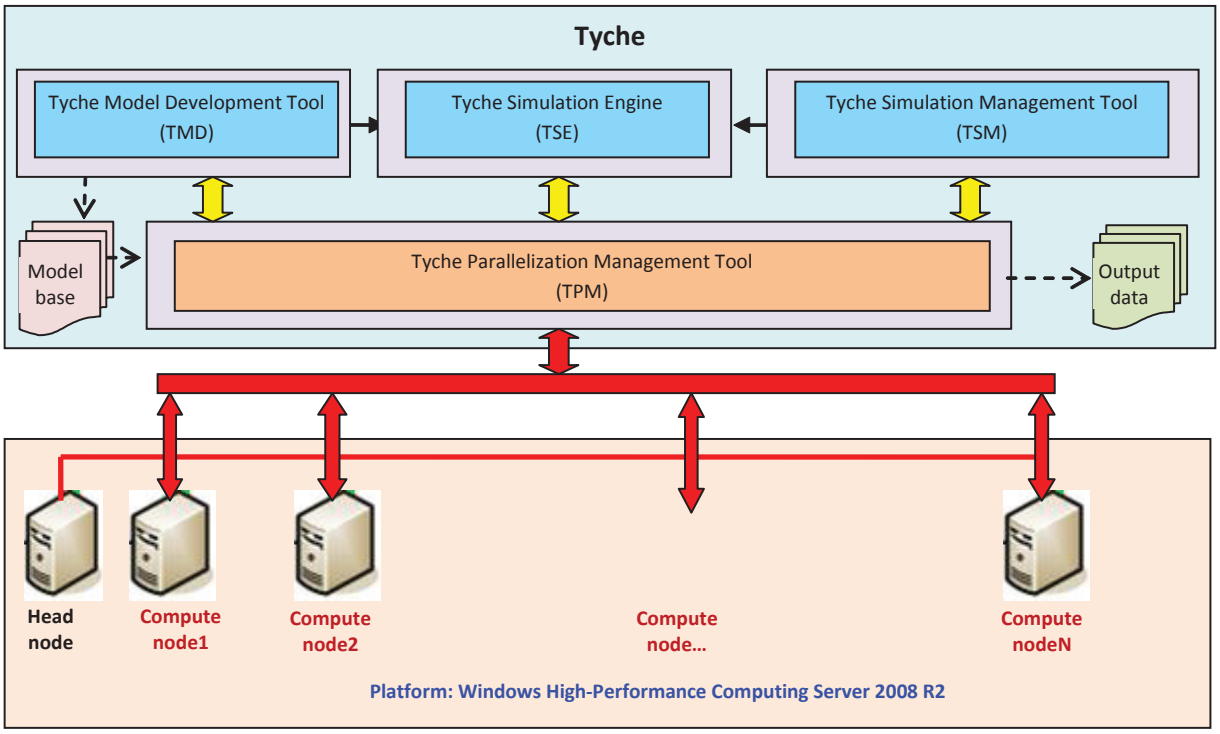

Figure 11: Proposed architecture for speeding up the Tyche simulation execution

4.2.2.2 Detailed Architecture with Simple Partitioning

Figure 12 shows a detailed view of the architecture described in Figure 11, in which each module is decomposed into several sub-modules.

The Tyche Model Development Tool (TMD), which is actually the current Tyche Editor GUI, consists of three sub-modules: the *Editor*, the *Tester* and the *Viewer*. The *Editor* is the main Graphical User Interface for developing force structure models visually, with all physical/conceptual entities and some procedural components, including capabilities, asset types, asset levels, bases, fleets, theatres, scenarios and phases, can be modelled and organized. The user force structure models created with the *Editor* can be saved as files in the Tyche Input File Format (*.tyi). With the *Tester* in TMD, which calls the Simulation Engine code, users are capable of testing force structure models with specified parameters including the input files, the numbers of iterations and the years to simulate. The *Viewer* in TMD is used to graphically examine the asset assignments made by the Tyche simulation engine after a simulation run. The associated assignment data are shown on a chart that mimics a fleet operational schedule.

The Tyche Simulation Engine (TSE) in the architecture is a backend module that conducts Monte-Carlo-based discrete event simulations. It consists of three sub-modules: *Pre-iterations*, *Iterations* and *Post-iterations*. The *Pre-Iterations* sub-module is used to initialize the system state variables, load force structure model files, and create various data structures used in simulation process. The *Iterations* sub-module is the core component of the simulation engine that implements all iterations in simulation instances. Its main functionality is to generate all possible events associated with a simulation instance, schedule events/assets, and form output result files. The *Post-Iterations* sub-module's purpose is to complete various statistical calculations for the analysis of military force structures. The current Simulation Engine in Tyche implements the sequential execution of iterations in a single simulation instance. In Section 4.3, a variety of ideas

and approaches will be suggested for a parallel and optimized simulation engine that could replace the current TSE in future.

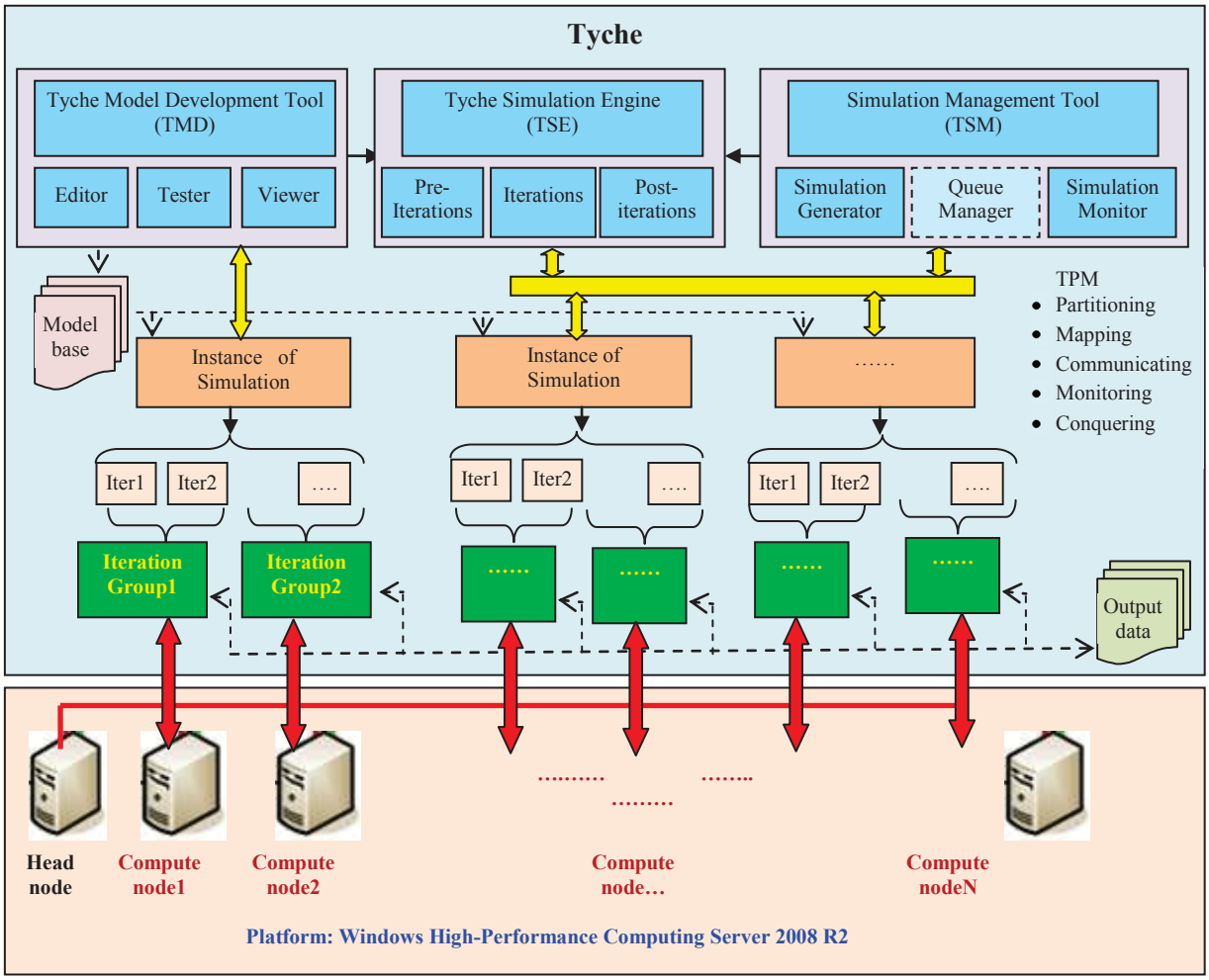

Figure 12: Detailed architecture with simple partitioning

The Tyche Simulation Management Tool (TSM) that relates to the current Tyche Dashboard directly is applied to generate and monitor multiple simulation instances that associate with a group of force structure models. The current Tyche Dashboard consists of two sub-modules, including the *Queue Manager* and the *Simulation Monitor*. The Tyche Dashboard calls the XML Editor to create multiple simulation instances, each of which is responsible for performing the simulation of a military force structure model with a large number of iterations. The *Queue Manager* is able to manage the execution order of simulation iterations. With the *Simulation Monitor*, users can monitor the status of simulation execution; start, pause, resume, and abort simulations; and clear completed simulations. In the new architecture, the Dashboard can be still used to generate simulation instances and monitor their execution, but the actual generation of simulation instances and queue management should be completed by the Tyche Parallelization Management Tool.

Compared to the current Tyche system, the Tyche Parallelization Management Tool (TPM) is a completely new module. It is responsible for partitioning, mapping, communicating, monitoring and conquering of parallel simulation execution. As mentioned in previous sections, each run of the Tyche system may consist of several simulation instances, and each simulation instance has a

large number of iterations. In most cases, the number of iterations is much larger than the number of cluster computer nodes. In the proposed architecture, TPM divides all iterations in simulation instances into approximately-balanced iteration groups, and then maps the iteration groups onto cluster computer nodes to be executed in parallel. Approximately-balanced partitioning refers to iterations in each simulation instance that are divided into several iteration groups and mapped onto cluster nodes. The size of each iteration group within a simulation instance is similar and approximately equals the overall average size of an iteration group in all simulation instances, but the sizes of iteration groups in different simulation instances may be different. A sample algorithm for approximately-balanced partition is given in Section 4.2.3.2.

The approximately-balanced partitioning is easy to implement, because it does not allow the iterations in a single group to come from different simulation instances. As a result, it is simple to handle multiple copies of an input file required by parallel execution and separated pieces of output file from parallel execution.

4.2.2.3 Detailed Architecture with Balanced Partitioning

Because the loading of cluster nodes is not balanced with iterations, the approximately-balanced partitioning described in Section 4.2.2.2 is not optimal. To achieve optimal loading, a balanced partitioning can be used. The idea for balanced partitioning is to divide all iterations in all simulation instances into equal-size iteration groups and map the groups onto cluster nodes, as shown in Figure 13. A sample algorithm for balanced partitioning is described in Section

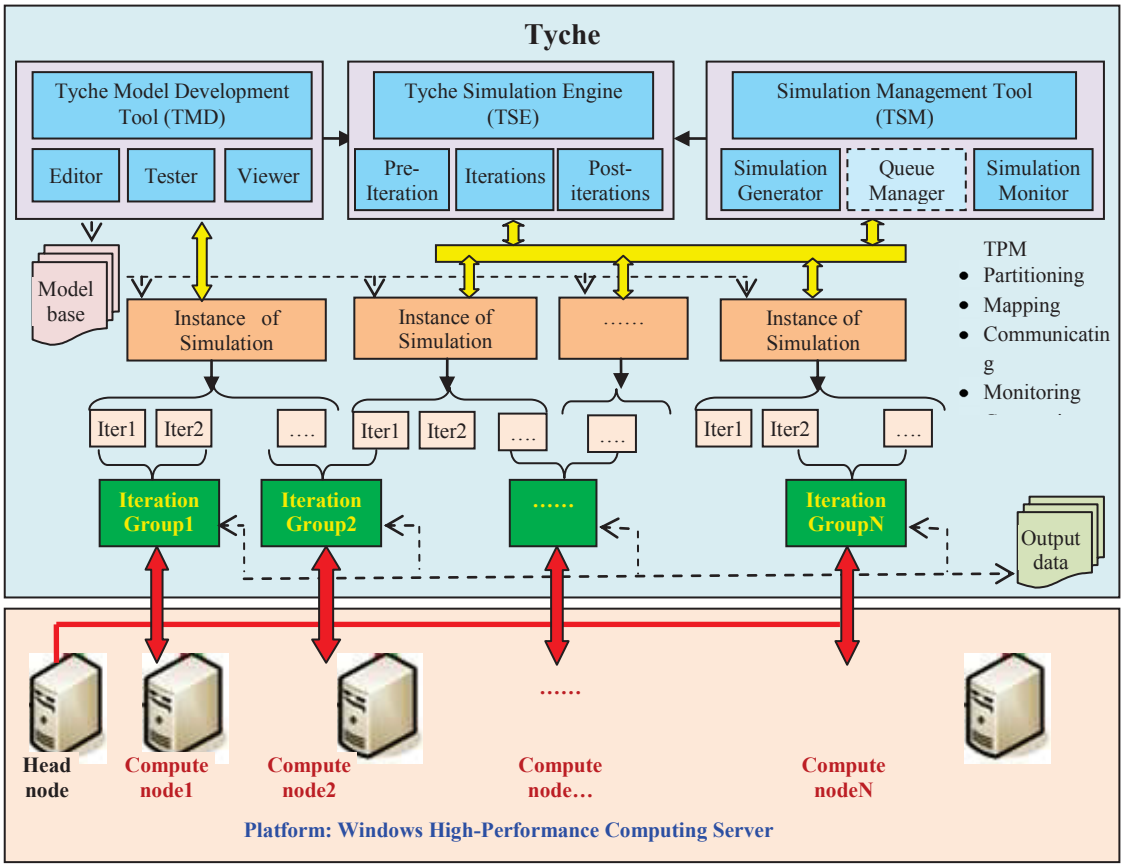

Figure 13: Detailed architecture with balanced partitioning

4.2.3.2. To pursue balanced loading, the iterations in a single group may come from two or more simulation instances. The main advantage of this partitioning is the balanced loading for all cluster nodes, and the main disadvantage is that additional efforts are required for handling multiple copies of input files required by parallel execution and some output files contain mixed data from different simulation instances.

4.2.2.4 Architecture with Two Levels of Parallel Execution

Multiple processors/cores in a cluster computer node on Windows HPC Server 2008 R2 make it possible to support two-level parallel execution of iterations in simulation instances.

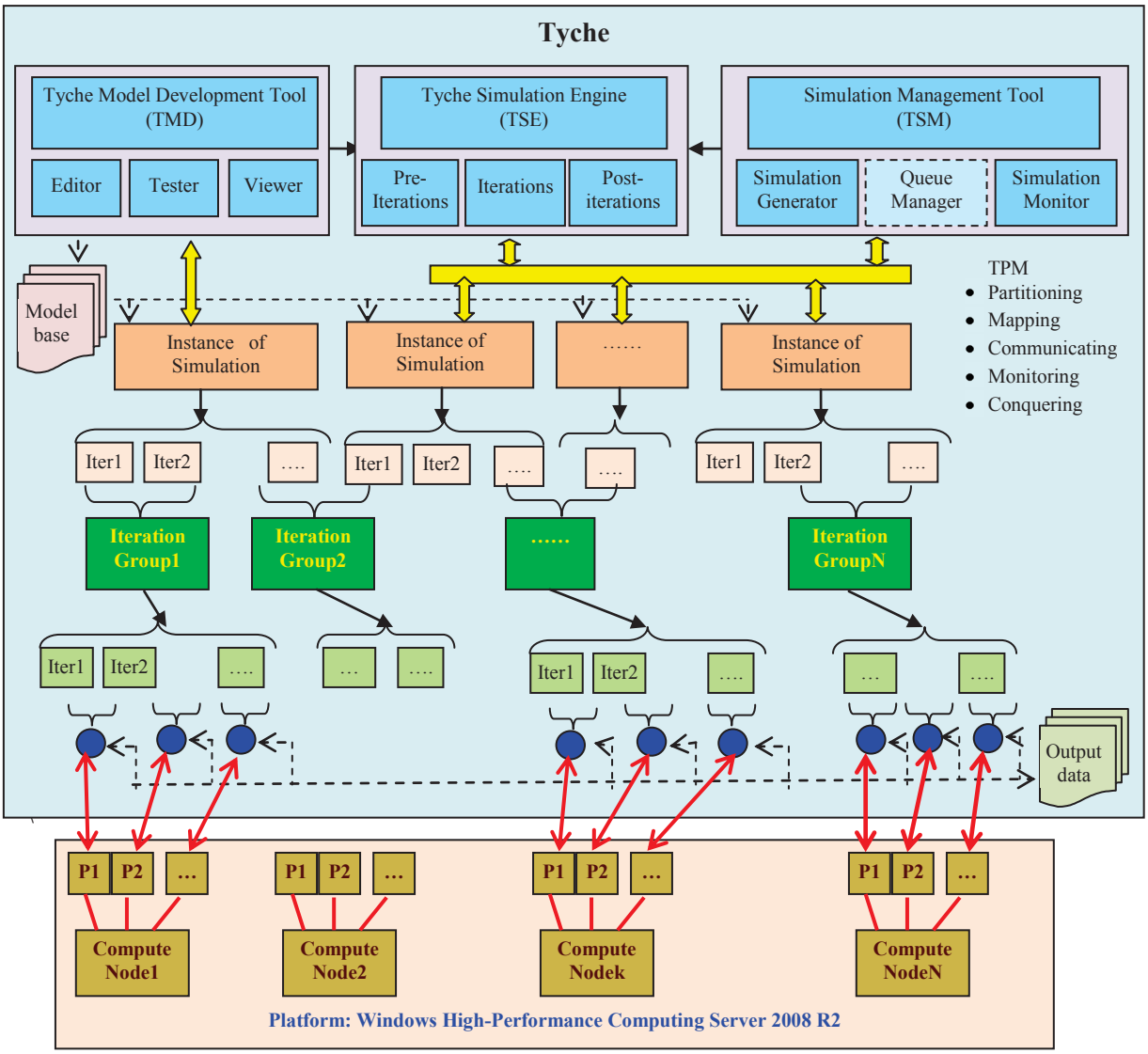

Figure 14: Architecture with two-level parallel execution

Figure 14 shows the updated architecture with two levels of parallel simulation execution. The Tyche Parallelization Management Tool first partitions all iterations in simulation instances into iteration groups, and then maps these iteration groups onto cluster nodes (green rectangles in Figure 14) . Furthermore, the Simulation Engine or the Tyche Parallelization Management Tool divides all the iterations within each iteration group into sub-groups(blue nodes in Figure 14),

each of which relates to a processor/core, and maps the sub-groups of iterations onto processors/cores to execute in parallel. The two-level partitioning and mapping makes full use of the parallel potential of Windows HPC Server 2008 R2 and can speed up the Tyche simulation execution further.

4.2.3 Partitioning and Mapping of Iterations in Simulation Instances

This section briefly reviews the principles and requirements of parallel programming, and then depicts two sample algorithms for partitioning iterations in simulation instances. Finally, it illustrates the approaches to map the Tyche modules onto jobs, tasks and computer nodes on a Windows HPC Sever 2008 R2.

4.2.3.1 Overview of Parallel Programming

Parallel programming utilizes *concurrency*, i.e. the ability to perform many actions simultaneously, to achieve high-performance computing (Sen, 2010; Foster 1995; Barney, 2010). Parallel programs are built by combining sequential programs. The goal is to allow independent sequential programs to run in parallel and produce partial results that then are merged into the final solution via patterns. *Concurrency*, *scalability*, *locality* and *modularity* are fundamental requirements for parallel software design (Foster, 1995).

Most programming issues may have several parallel solutions. The best solution may differ from that suggested by existing sequential algorithms. One of the most popular design methodologies for parallel computing, proposed by Foster (Foster, 1995), is intended to foster an exploratory approach to design in which machine-independent issues such as concurrency are considered early and machine-specific aspects of design are delayed until late in the design process. This methodology structures the design process as four distinct stages: partitioning, communication, agglomeration, and mapping (PCAM). In the first two stages, it focuses on concurrency and scalability and seeks to discover algorithms with these qualities. In the third and fourth stages, attention shifts to locality and other performance-related issues. The four stages can be summarized as follows:

- *Partitioning*. The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.

- *Communication*. The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.

- *Agglomeration*. The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.

- *Mapping*. Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

4.2.3.2 Algorithms for Partitioning Iterations in Simulation Instances

This section describes two sample algorithms that are used to implement the approximately-balanced and balanced partitioning mentioned in the previous Sections 4.2.2.2 and 4.2.2.3.

4.2.3.2.1 Sample Algorithm for Approximately-Balanced Partitioning

The basic idea of approximately-balanced partitioning is to divide the iterations in a simulation instance into iteration groups and map the groups onto cluster nodes. The ideal average size of iteration group is equal to the number of all iterations in simulation instances divided by the number of cluster computer nodes. To keep things simple, all iterations in each iteration group are limited to a single simulation instance. In other words, the iteration groups belonging to a single simulation instance have similar sizes, but the iteration groups from different simulation instances may have different sizes. As a result, the actual number of iterations in each iteration group is approximately equal to the ideal average size of an iteration group.

For example, there are two simulation instances with 1000 and 2100 iterations respectively, which are executed on a cluster with four computer nodes. A possible approximately-balanced partitioning is to assign the 1000 iterations in the first simulation instance to one computer node and the 2100 iterations in the second simulation instance to three remaining computer nodes, each of which executes 700 iterations.

Assume that the number of cluster nodes is N_c, the number of simulation instances in a Tyche running is N_S, the number of iterations in the i-th simulation instance is N_{Si}, and the total number of iterations in all simulation instances is N_{tot}. Figure 15 describes the sample algorithm in detailed steps.

Algorithm: SampleApproximatelyBalancedPartitioning
Parameters:

- S: set of simulation instances
- N_c: number of cluster nodes
- N_s: number of simulation instances

Procedure:

1. Variables:
   - N_{tot}: total number of iterations
   - A: overall average number of iterations in each iteration group based on all simulation instances
   - S_i: the i-th simulation instance in S
   - N_{si}: the number of iterations in S_i
   - N_{sci}: *the number of iteration* groups for S_i
   - T: a tree structure for iteration groups, in which each simulation instance S_i has a sub-tree T_i; each sub-tree contains one or more iteration groups associated with the simulation instance, and the number of iterations in each group is approximately equal to the overall average number of iterations.
2. Compute N_{tot} <- sum of all N_{si} (i = 1, 2, …, N_s)
3. Compute A <- N_{tot} / N_c
4. Create a *Partitioning Tree*, T, to hold the result of partitioning of simulation instances
5. Dividing:
   a. Check each simulation instance, S_i, in S:
      i. If S_i is the last simulation instance, then partition all iterations in the simulation instance into groups that correspond to all remaining cluster nodes , otherwise do the following ii to iv;
      ii. Assign the number of iterations in S_i *to* N_{si}
      iii. Compute the number of iteration groups for S_i: N_{sci} <- (int) (N_{si} / A)
      iv. Build a partitioning sub-tree for S_i, called T_i, in which the root of the sub-tree is the simulation instance S_i, and all sub-nodes under the root are iteration groups in this simulation instance, and the number of iterations in each group is equal to N_{si}/ N_{sci}
6. Mapping: Assign each *iteration group in the Partitioning Tree* onto a cluster node.
7. Return the result of partitioning and mapping

Figure 15: Sample algorithm for approximately-balanced partitioning

4.2.3.2.2 Sample Algorithm for Balanced Partitioning

The balanced partitioning is to optimize the work load of cluster nodes by keeping all iteration groups equal in sizes.

This balanced partitioning algorithm examines all simulation instances and tries to divide all iterations into N_c groups, each of which has a relatively equal number of iterations for a computer node.

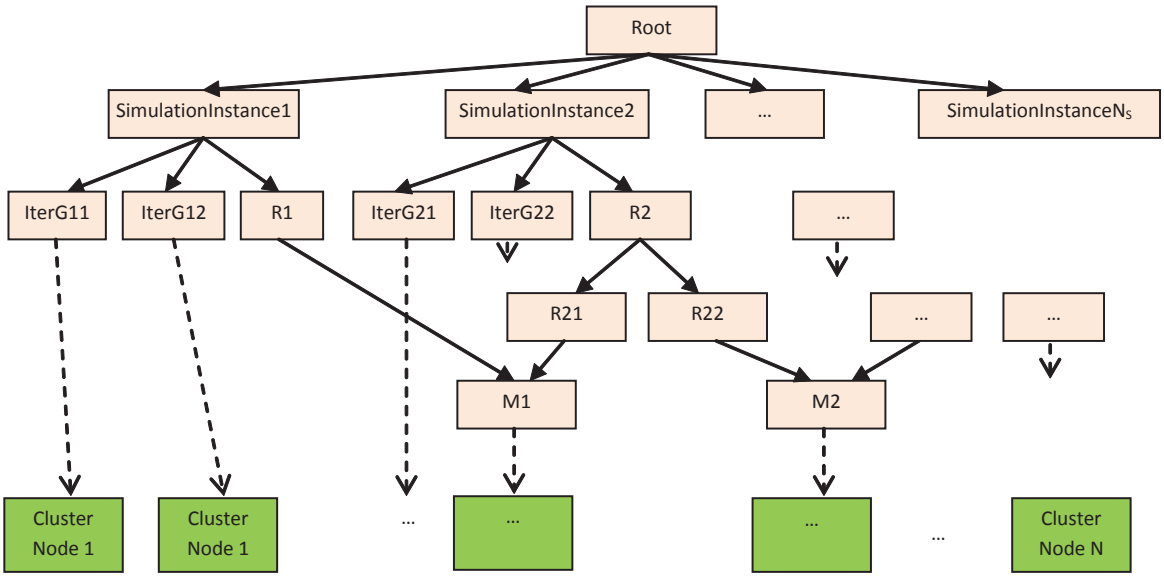

Figure 16: Partitioning and mapping of simulations and iterations

As a result, compared to the average number of iterations in each iteration group, each simulation instance may contain an iteration group that has less iterations than the average number of iterations in each iteration group. For example, in Figure 16, assume that the average number of iterations in each group is 900, and the *SimulationInstance2* has 2000 iterations. The *SimulationInstance2* is divided into three groups: *IterG21* with 900 iterations, *IterG22* with 900 iterations and *R2* with 200 iterations. All the *IterGxx* groups in Figure 16 are known as *Terminal Groups*, and all the *Rx* groups in Figure 16 are called as *Remaining Groups*. This algorithm then tries to re-organize all the *Remaining Groups* into *Mixed Groups* with the average number of iterations in each group. However, to keep each *Mixed Group* having the average size of iteration group, further partitioning may be required for some *Remaining Groups*. For example, in Figure 16, the *Remaining Group* of *SimulationInstance2*, R2, may be further divided into two sub-groups, *R21* and *R22*, for the *Mixed Groups M1* and *M2* respectively to meet the requirements of balanced loading.

The result of such partitioning is a partitioning graph as shown in Figure 16, and all leaf nodes in the graph are the finalized iteration groups for parallel mapping and execution.

Algorithm: SampleBalancedPartitioning
Parameters:

- S: set of simulations
- N_c: number of cluster nodes
- N_s: number of simulation instances

Procedure:

1. Variables:
   - N_{tot}: total number of iterations
   - A: average number of iterations in each iteration group
   - S_i: the i-th simulation in S
   - N_{si}: the number of iterations in S_i
   - N_{ri}: the number of remaining iterations: N_{si} mod A
   - T: a graph structure for iteration groups, in which each simulation instance S_i has a sub-graph T_i; in T_i, the iterations of S_i are divided into groups with the average numbers of iterations; there may be one group in each simulation instance whose number of iterations is less than the average number of iterations in each cluster node, called *RemainingGroup*, and all other groups, called *TerminalGroups*, have the average numbers of iterations
2. Compute N_{tot} <- sum of all N_{si} (i = 1, 2, …, N_s)
3. Compute A <- N_{tot} / N_c
4. Create a *PartitioningGraph*, T, to hold all iteration groups of simulation instances
5. Dividing:
   a. Check each simulation instance, S_i, in S:
      i. Assign the number of iterations in S_i *to* N_{si}
      ii. Build a sub-graph of partitioning for S_i, called T_i, the root of T_i is S_i
      iii. Divide the iterations in S_i into the groups with average size of iterations; add each iteration group to T_i as a sub-node; all the groups with average size of iterations are TerminalGroups; if there is a group with less iterations than the average size of iterations, it is a *RemainingGroup*.
6. Merging/Re-organizing *RemainingGroups*:
   a. Combine multiple *RemainingGroups* into *MixedGroups,* each of which has the average size of iterations; to get balanced *MixedGroups,* dividing some *RemainingGroups* into sub-groups may be required in this step.
7. Mapping: Assign each *TerminalGroup* or *MixedGroups* in *PartitioningGroup* onto a cluster node.
8. Return the result of partitioning and mapping.

Figure 17: Sample algorithm for balanced partitioning

Finally, the algorithm maps all the finalized iteration groups onto cluster nodes to execute in parallel. Figure 17 describes the detailed steps of balanced partitioning.

Please note that both the approximately-balanced and balanced algorithms omit the directions for handling input and output files. In fact, to meet the needs of parallel execution, depending on the implementation of the simulation engine, multiple copies of an input file or data structures of a force structure model may be required. In addition, the output data file related to each simulation instance may be separated into multiple pieces owing to the parallel partitioning and execution. The Tyche software system should also merge related output data pieces to form the final output data file for each simulation instance.

Furthermore, both algorithms ignore the number of simulated years in individual simulation instances, which may or may not be equal. This also affects the work loading of cluster nodes. In future Tyche development, this factor may be considered further.

4.2.3.3 Mapping Tyche Modules onto Jobs, Tasks and Computer Nodes on Windows HPC Server 2008

To make the Tyche system compatible with Windows HPC Server 2008, the mapping of components between them have to be taken into account. The Job Manager or Scheduler in Windows HPC Server 2008 is very flexible for users, and most applications can be mapped onto either jobs or tasks in Windows HPC Server 2008 for parallel execution. Based on the proposed architectures in Section 4.2.2.1 – 4.2.2.3, this section describes a group of sample mapping between the Tyche system and the Windows HPC Server 2008.

4.2.3.3.1 Single-level Parallel Mapping

A single-level parallel mapping refers to the association between the Tyche modules/functionality and the jobs/tasks in Windows HPC Server 2008 for cluster nodes. The following Table 1 shows a sample mapping.

Table 1: Single-level parallel mapping

Module/functionality in Tyche	Components in Windows HPC Server 2008 R2
Tyche Model Development Tool (TMD or Tyche GUI)	Job
Tyche Simulation Management Tool (TSM, or Dashboard)	Job or Job Scheduler API
Tyche Parallelization Management Tool (TPM)	Job or Job Scheduler API
Tyche Simulation Engine (TSE)	A task in job
Tyche simulation instance	Jobs or tasks
Tyche iteration group	A job or task
Tyche iteration	A call to the *DoIteration* function in the simulation engine

The use of an example helps to explain the ideas of single-level parallel mapping. Assume that the Parallelization Management Tool is generating and managing the following three simulation instances on a platform with $_{\text{eight}}$ computer nodes, each of which has 12 cores:

- *Sim#1* including 3000 iterations for seven simulated years with input file IF#1,
- *Sim#2* including 1000 iterations for five simulated years with input file IF#2, and

- *Sim#3* including 2500 iterations for six simulated years with input file IF#3.

The following describes alternative sample methods for the partitioning and mapping of iterations in the simulation instances. There are 6500 iterations in total in the three simulation instances, and the average number of iterations on each cluster node is 813.

- Method #1:
    - If adopting approximately-balanced partitioning, TPM may uses four computer nodes for *Sim#1*, one computer node for *Sim#2*, and three computer nodes for *Sim#3* respectively;
    - A job is created for the TPM, which consists of eight parametric sweep tasks:
        - Task#1 - Task#4: dividing 3000 iterations in *Sim#1* into four tasks, each of which is a call to the Simulation Engine with 750 iterations for seven years, and mapping the four tasks onto the cluster node #1 to #4;
        - Task#5: A task for *Sim#2* with 1000 iterations for five years, mapped onto the cluster node #5;
        - Task#6 - 8: dividing 2500 iterations in *Sim#3* into three tasks, each of which is a call to the Simulation Engine with 833, 833 & 834 iterations respectively for six years, and mapping the three tasks onto the cluster node #6 to #8;
    - Execute all the tasks in parallel;
    - TPM then merges all separated output files into final output files associated with each simulation instance;
    - TPM handles statistical analysis of the output files.
- Method #2:
    - This method assumes that the Windows HPC Server Job Scheduler API is used in the Parallelization Management Tool to partition iterations and map them onto cluster nodes;
    - The TPM creates a job for each simulation instance as follows:
        - Job#1: containing four tasks for 3000 iterations in *Sim#1*, each of which is a call to Simulation Engine with 750 iterations and seven years to simulate, mapped onto the cluster node #1 to #4;
        - Job#2: comprising a task with 1000 iterations for five years in *Sim#2*, and mapped onto the cluster node #5;
        - Job#3: including three tasks for 2500 iterations in *Sim#3*, each of which is a call to Simulation Engine with 833, 833 & 834 iterations respectively for six years, mapped onto the cluster node #6 - #8;
    - Execute all the jobs and tasks in parallel;
    - TPM then merges all separated output files into final output files associated with each simulation instance;
    - TPM handles statistical analysis of the output files.
- Other Methods:
    - Jobs and tasks in Windows HPC Server 2008 are very flexible for users to complete their work. For example, in the above Method#2, the TPM may also divide all iterations in three simulation instances into eight jobs, each of which contains a task calling TSE with 750 ~1000 iterations for the associated years to simulate.

4.2.3.3.2 Two-level Parallel Mapping

If each cluster computer node consists of multiple processors or cores, the parallel mapping of iterations can be characterized at two levels: i.e. the cluster-node level and the core level. Table $_{two}$ illustrates a sample mapping with two-level parallelization between the Tyche modules and components of Windows HPC Server 2008 R2.

Table 2: Two-level parallel mapping

Module/functionality in Tyche	Components in Windows HPC Server 2008 R2
Tyche Model Development Tool (TMD or TycheGUI)	Job
Tyche Simulation Management Tool (TSM, or Dashboard)	Job
Tyche Parallelization Management Tool (TPM)	Job or Job Scheduler API
Tyche Simulation Engine (TSE)	A task in job
Tyche simulation instance	Jobs or tasks
Tyche iteration group	A job or task
Tyche sub-iteration group	A call to the *DoIterationGroup* function in TSE to handle a sub-group of iterations (used for parallel execution on multi-core/processors)
Tyche iteration	A *DoIteration* function call in the simulation engine

The main difference between the single-level and two-level parallel mappings lies in the implementation means of the simulation engine. If the implementation of the simulation engine supports the partitioning of each iteration group associated with a cluster node into sub-groups and mapping the sub-groups onto multi-processors/cores, the creation and management of two-level jobs and tasks on Windows HPC Server 2008 are simply to use jobs and tasks to call the parallel simulation engine. The Simulation Engine with such parallelization capability is very powerful and flexible for the future Tyche system. However, if the Simulation Engine has no parallel capability, the two-level parallelization can also be implemented with jobs and tasks on Windows HPC Server 2008, but many manual efforts may be required for handling input and output files.

4.2.3.3.3 Supporting Scalability

Scalability in parallel computing refers to the adaptability of a software system for increasing/reducing computer nodes in a parallel platform. Different platforms may contain

different numbers of computer nodes, and for a given platform, the number of available computer nodes may also change from day to day. How does the new architecture handle such scalability? Actually, there are alternative means to support varied system configurations. For example, depending on the implementation of Tyche Parallelization Management Tool and Simulation Engine, a property file may be used for users to change system configurations including the number of available computer nodes or processors in each computer node. In addition, the command line parameters for the numbers of available computer nodes and cores can also be used for supporting flexible platform configurations.

4.3 Optimizing the Design of the Simulation Engine

One of objectives in this project is to improve the design of the simulation engine to speed up simulation execution further. This section describes the approach to achieve this objective with optimized and parallelized functions within the simulation engine.

4.3.1 Optimizing *Iterations* to Speed up Simulation

This section first analyzes the main functions used in the *Iterations* step that is the most time-consuming procedure in the simulation engine, including the *ResetAssetData*, *GenerateEvents*, *ScheduleAssets* and *OutputData* functions. It then proposes a group of optimization approaches for the future software design of the simulation engine to speed up simulation execution further.

4.3.1.1 Optimizing the *GenerateEvents* Function in Iteration

The *GenerateEvents* function is used to generate all possible events for all the phases and asset levels with "Schedule" or "Random" types, based on the specified years to simulate and frequency for a given simulation instance.

The basic idea to improve the performance of *GenerateEvents* function is to identify unchanged and changed components from iteration to iteration, and then try to move the unchanged components out of the iteration and keep the changed components inside the iteration.

With the analysis of source code in *GenerateEvents* function, the following data and variables are identified as unchanged components from iteration to iteration in a given simulation instance:

- the user force structure model,
- the types of asset levels,
- the types of phases of scenarios,
- the number of simulated years,
- the number of iterations, and
- the basic relationships between events and asset levels/phases.

All iterations in a given simulation instance use the same force structure model, so that all the basic relationships between events and asset levels/phases are not changed from iteration to iteration.

The changed components between iterations include the start dates, offsets and durations of events/asset levels.

The suggested solution for the *GenerateEvents* function is to divide it into two functions, *GenerateBasicEvents* and *AdjustEventTime*, as shown in Figure 18. The *GenerateBasicEvents* function uses the force structure model to generate all possible events for all the asset levels and scenario phases with "Scheduled" type, with a temporary start date of the iteration. The *AdjustEventTime* function is to generate random events and re-compute the start dates and durations of each event with the actual start date of the current iteration. With the re-design of the simulation engine, the *GenerateBasicEvents* function is called before the *DoIteration* function, and the *AdjustEventTime* function is called in the same place as the previous *GenerateEvents* function within the *DoIteration* function. Like the asset list in the current Tyche software system, the basic event list could be shared by all iterations.

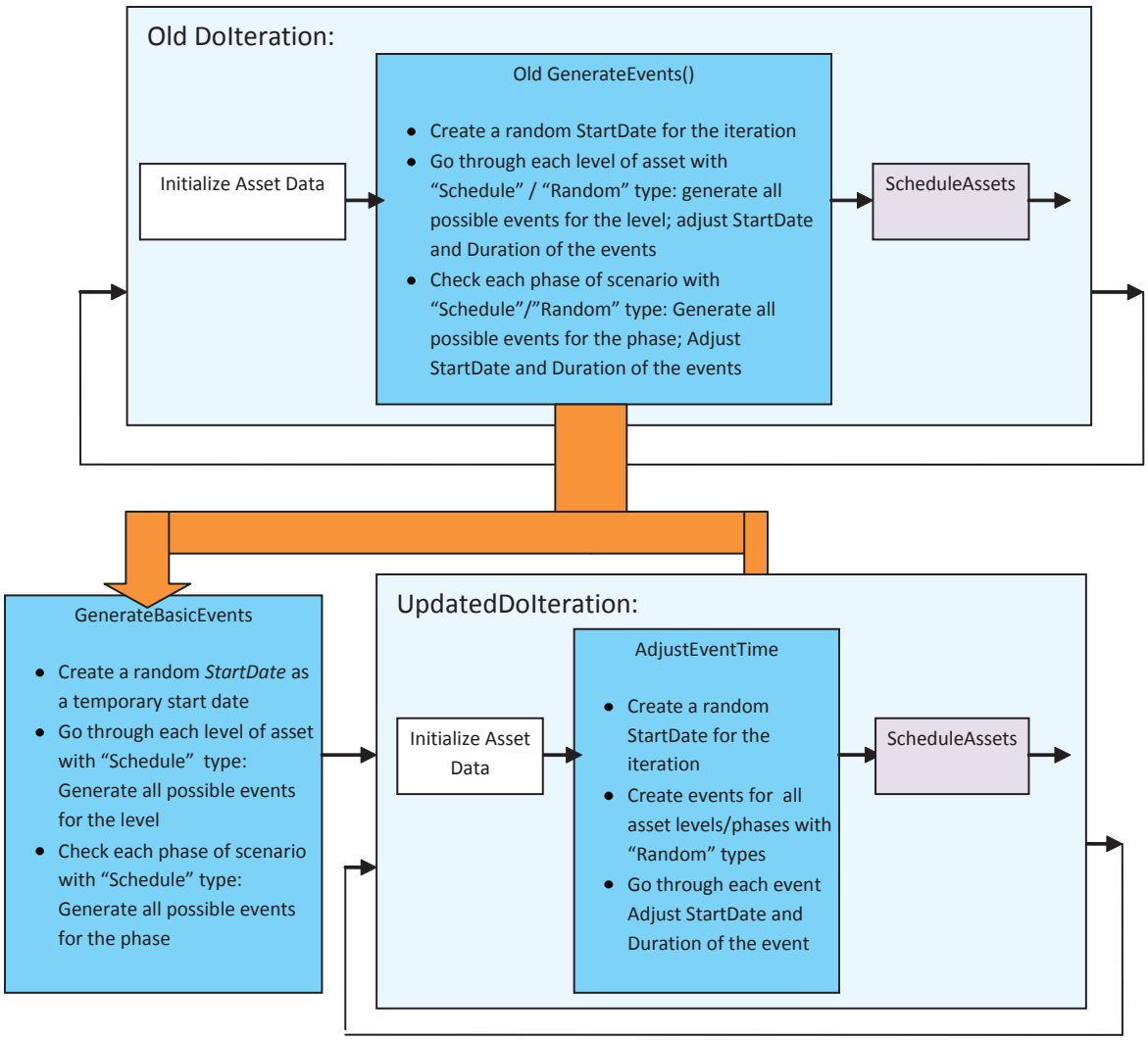

Figure 18: Optimizing GenerateEvents function in iteration

4.3.1.2 Optimizing the *ScheduleAssets* Function in Iteration

The *ScheduleAssets* function is the core procedure of the Tyche simulation engine. Its main functionality is as follows:

- Setting *currentDate* to 0;

- Going through each event in the system for following operations:
    - Advancing the *currentDate* with the event's *StartDate*;
    - Searching demanded and available assets for the event;
    - Computing the departure and arrival time of assigned assets;
    - Checking the bump and the re-schedule policy of event for possible new event generation.

As described in the previous sections, all iterations in a given simulation instance share the same force structure model. Therefore, the capability dependency relationships between assets do not change from iteration to iteration. As a result, the "Search demanded assets for each event" operation could be moved to the front of *DoIteraiton* function in the simulation engine, and form a candidate demanded (pre-scored) asset list. In the updated *ScheduleAssets* function within iteration, it checks the availability of each candidate demanded asset and assigns available assets for each event.

Figure 19 summarizes the ideas to optimize the *ScheduleAssets* function. A new sub-function, called *PreSearchAssets*, is defined to perform the functionality of "Search demanded assets", and generate a candidate asset list for each event. The *PreSearchAssets* sub-function is called by the simulation engine before the *DoIteration* function. All other functionality in the previous *ScheduleAssets* function is not changed, and the updated *ScheduleAssets* function is kept within the *DoIteration* function.

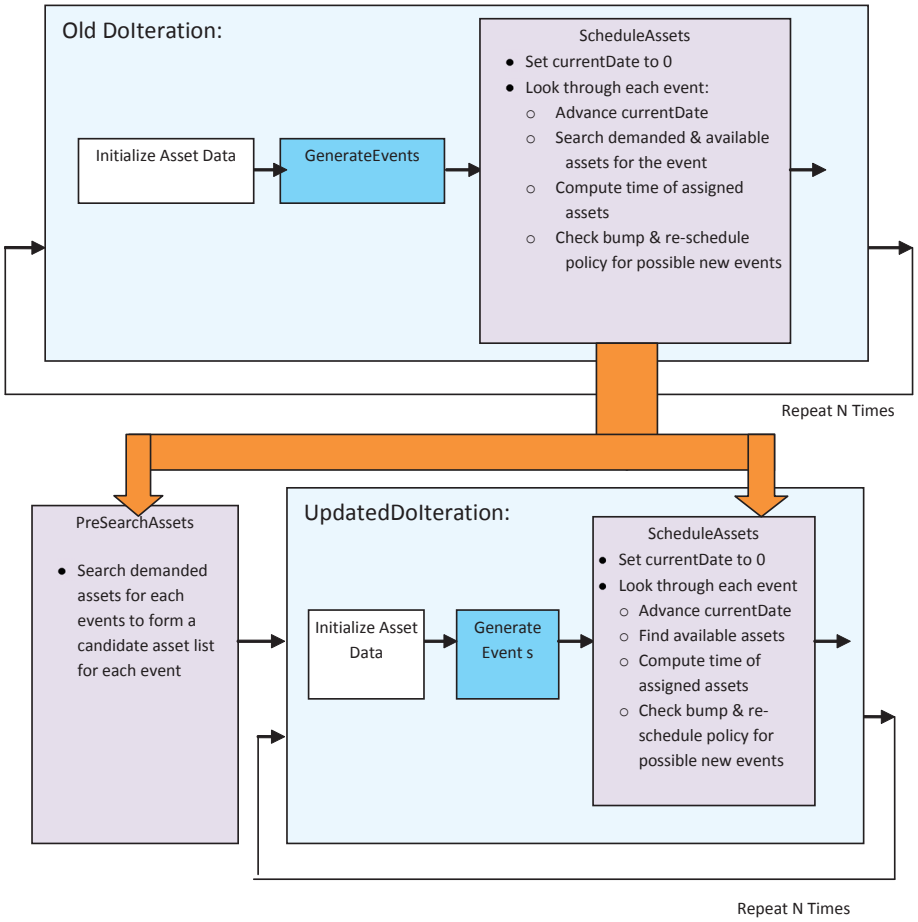

Figure 19: Optimizing ScheduleAssets function in iteration

4.3.1.3 Optimizing the *OutputData* Function in Iteration

The *OutputData* function called in iteration is mainly used to generate and write result messages to output files. Specifically, it extracts and writes the simulation-instance-related information first, such as the number of iterations and the number of simulated years. Then, it outputs the names of data columns for events, including source type, source name, cause name, start date, end date, location, re-schedule activity, and all assigned assets with attributes, such as asset name, level names, base departure date, theatre arrival date, theatre departure date, and base arrival date. There is also a flag for the bumped indicator of associated asset level written into the output file. The current Tyche system accesses the file stream once for each event. Owing to a large number of events in a typical simulation iteration, the file output operations are time consuming. The suggested solutions here (as shown in Figure 20) include reducing file output operations, encoding text messages and using multiple threads. If the output information of multiple events is merged into a buffer for writing, the file access operations may be reduced. However, with this approach, some result data may be lost when errors occur in simulation process. For example, if the buffer size is enlarged to hold the output messages of 50 events, in the worst case, if the system encounters an error just before the full-size buffer is written to file, then the result data of the last 50 events may be lost. Another way is to use multiple-threads to reduce the time spent for the output file within an iteration.

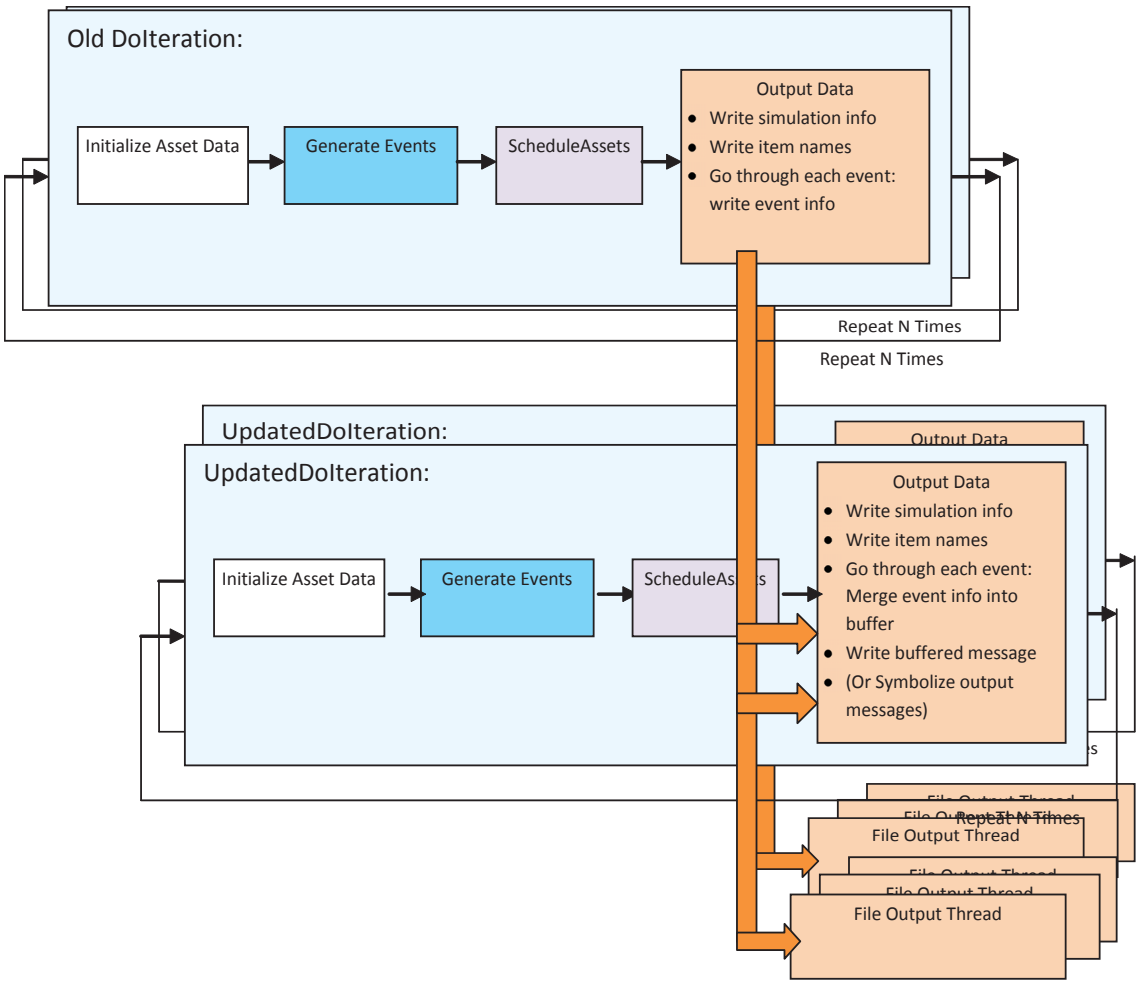

Figure 20: Optimizing OutputData function in iteration

In addition, some simple compression or encoding techniques may also be used to reduce the sizes of output files, e.g. using numbers or abbreviations to represent names/text used in event messages. For example, if using the numbers, 1, 2, and three to represent the texts "Source Type", "Source Name", and "Maritime Coastal Defence Vessel (MSDV)", the output messages will be compressed to a great degree. The disadvantage of encoding is that the output then becomes machine readable only, as the user is not able to look at the contents without reference to further information.

4.3.1.4 Summary of Optimized *Iterations* in the Simulation Engine

In summary, by combining the results from the Sections 4.3.1.1 to 4.3.1.4, an optimized core step, *UpdatedIterations*, is formed, as shown in Figure 21. It reduces the redundant or repeated operations from iteration to iteration to speed up simulation execution further.

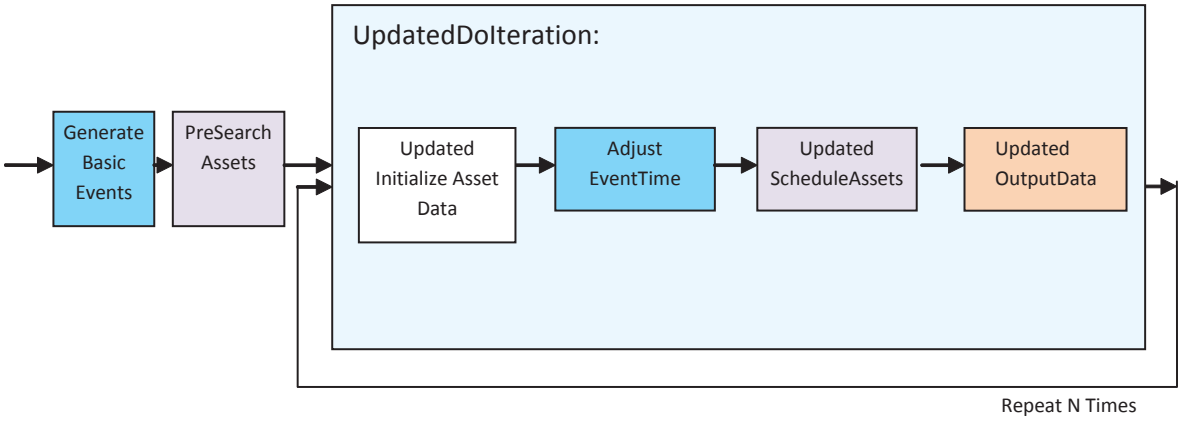

Figure 21: Optimized DoIteration in the simulation engine

4.3.2 Using Parallel Functions to Speed up Simulation

Usually, the number of iterations (for example 1000) in several Tyche simulation instances, is large enough for two-level parallel partitioning and mapping because it is much larger than the number of cluster nodes and cores in each cluster node. However, when the number of iterations of a Tyche simulation instance is comparable to the number of cluster nodes, simulation can be further sped up by dividing the functions in iterations into parallel segments. Figure 22 shows the basic ideas for parallel functions within iterations, in which each iteration consists of a group of parallel functions to be executed in parallel on processors/cores. The following briefly describes the ideas to convert some main functions in the simulation engine into parallel segments.

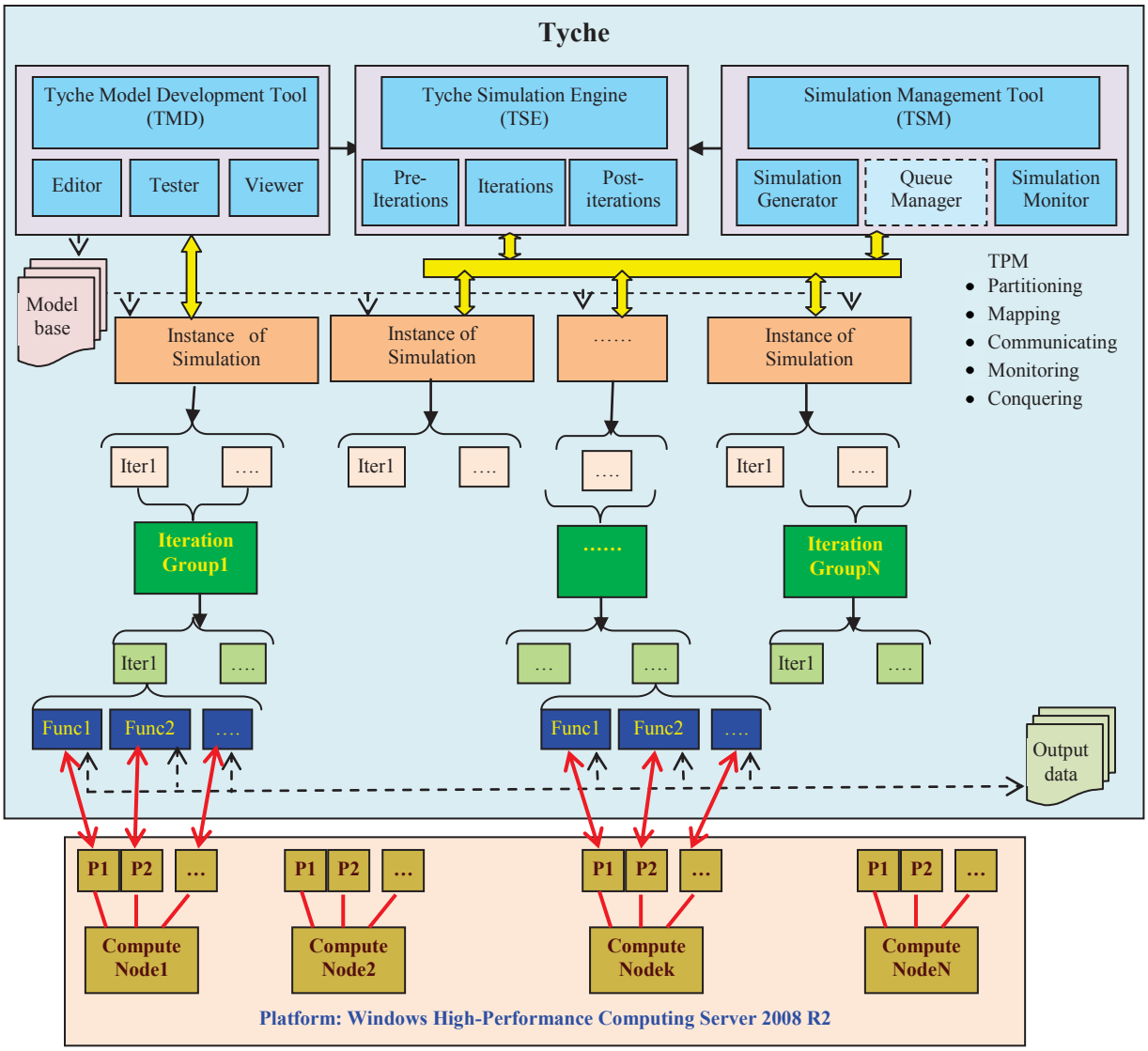

Figure 22: Using parallel functions to speed up simulation

4.3.2.1 Parallelizing Functions used in *Pre-iterations*

The main functions in pre-iterations consist of the *BuildAssets* and *BuildScenarios* functions. In the Tyche system, all assets and scenarios are independent entities. Therefore, they can be created in parallel. The following list includes the partitioned segments for the two functions that can be executed in parallel.

- Parallel segments in *BuildAssets* function:
  - *BuildInternalAssets* with sub-parallel-segments:
    - Create InternalAsset1
    - Create InternalAsset2
    - …
  - *BuildExternalAssets* with sub-parallel-segments:

        - Create ExternalAsset1
        - Create ExternalAsset2
        - …
- Parallel segments in *BuildScenarios* function
    - Generate Scenario1
    - Generate Scenario2
    - …

4.3.2.2 Parallelizing Functions used in *Iterations*

Parallelized functions can also be used within a single iteration. As mentioned in previous sections, there are four main functions used in *Iterations*, including *ResetAssetData*, *GenerateEvents*, *ScheduleAssets* and *OutputData*. The following list describes some possible parallel segments for the functions:

- Parallel segments in *ResetAssetData* function:
    - Reset data of Asset1
    - Reset data of Asset2
    - …
- Parallel segments in *GenerateEvents* function:
    - Generate events for Asset1
    - Generate events for Asset2
    - …
    - Generate events for Scenario1
    - Generate events for Scenario2
    - …
- Parallel segments in *ScheduleAssets* function:
    - UpdateActivity for Asset1
    - UpdateActivity for Asset2
    - …
        - Search candidate assets for Event1
        - Search candidate assets for Event2
        - …
            - Select assigned assets for Event1
            - Select assigned assets for Event2
            - …

4.3.2.3 Parallelizing Functions used in *Post-iterations*

The main function in *Post-iterations* is the *GenerateStatistics* function that calls other functions including *OutputAssetStatistics*, *OutputScenarioStatistics*, *OutputCapabilityStatistics*, and *OutputRisk*. In fact, these functions can also be executed in parallel.

4.4 Evaluating and Recommending Programming Environments

A good programming environment can help develop a more powerful and reliable software system easily and quickly. The following methodology is used for evaluating and recommending programming environments for future Tyche development:

- Establishing criteria for assessment of tools;
- Collecting survey/review/summarization papers or documents about various tools for discrete event simulation;
- Evaluating and comparing tools;
- Recommending a group of alternative programming environments for future Tyche development.

4.4.1 Establishing Criteria for Evaluating Programming Environments

Based on the current Tyche system and the objectives of the future Tyche system, the following criteria were established by DRDC CORA and the project team, which were used to evaluate the quality of programming environments:

- Mandatory requirements:
    - Compatibility with the current Tyche GUI for data entry and post-processing,
    - Compatibility with the current Tyche Dashboard for running on a standalone desktop,
    - Compatibility with Windows HPC Server 2008 R2 and ability to communicate with HPC Job Manager via standard I/O, error channels using command line, prompt) – this enables parallel processing at the job level,
    - Increased speed of code execution (the faster, the better!), and
    - Support for complex programming using common language(s),
- Requirements ranked in order of preference:
    - Parallel capability to task/programming level,
    - Ease of use and maintenance,
    - Domain adaptability, i.e. suitable for force structure analysis or at least for generic simulation,
    - Open source (to minimize cost),
    - Popularity, and
    - Platform independency.

4.4.2 Gathering information about programming environments for DES/PDES

The main source of information about programming environments was the Internet. Various keyword combinations were used to search the Internet for the surveys, reviews and summarizations of libraries, languages, Application Programming Interfaces (APIs), frameworks,

software or tools for discrete event simulation or parallel discrete event simulation. As a result, 22 papers/lists/documents of survey/review/summarization of DES or PDES are identified, which covers 350 tools for DES or PDES (See Appendix A for the list of tool names). The main information sources are as follows: Altmann, 2011; Barr, et al., 2004; Buyya & Murshed 2002; CACI, 2011a, 2011b, 2011c; Cambridge SRG, 2011; Constitution, 2011; Filippi, et al.,2011; Goble, 1997, Goes, et al, 2011a, 2011b; Gomes, 2011; Horejsi, 2011; Isee systems, 2011a, 2011b, 2011c; Janousek & Kironsky, 2006; Lanner, 2011; Lantz, 2008; Lee & Neuendorffer, 2007; Low, et al., 1999; Markt & Mayer, 1997; Mathworks, 2011a, 2011b; McGregor & Cain, 2011; Mottelet & Pauss, 2007; Pegden & Sturrock, 2010; Pegden, 2011a, 2011b; Ptolemy, 2011; Rizzoli, 2011; Rockwell Automation, 2011; Saker Solutions, 2011; Sen, 2008; Simio, 2011a, 2011b, 2011c; Simul8, 2011a, 2011b; Sulistio, et al., 2011, 2004; Thuyet, 2010; Trumphurst, 2011; Wainer, 2011; Wikipedia, 2011a, 2011b, 2011c, 2011d, 2011e; Xj Technologies, 2010, 2011; Yong,2011.

4.4.3 Evaluating Programming Environments

Evaluating, comparing and sorting 350 tools was beyond the scope of this work. As the first step of evaluation, the number of citations of each tool in all the survey/review/summarization papers and documents reviewed was used generating the list found in Appendix A. The project then sorted the citation frequency, and took the top 42 tools with the most citations for further evaluation, as shown in Table 3.[1]

Table 3: Top 42 + 1 tools with the most citations

AMESim (4)	AnyLogic (6)	Arena (8)	AutoMod (5)
C++SIM (5)	CSIM17/19 (4)	Dymola (4)	ExtendSim (4)
Flexsim (6)	Galatea (5)	GoldSim (6)	GPSS/H (8)
GridSim (4)	JDEVS (4)	JiST (4)	MODSIM III (4)
NetSim (5)	NS2/NS3 (4)	ParaSol (4)	Parsec (5)
Plant Simulation (5)	ProModel (4)	Ptolemy (5)	Scicos (4)
Scilab (4)	SeSAm (4)	SimCAD Pro (4)	Simjava (5)
Simio (5)	Simkit (6)	SimPy (8)	SIMSCRIPT II.5, III (7)
SIMUL8 (6)	Simula (5)	Simulink (4)	SmallDEVS (4)
STELLA II (4)	Vensim (5)	VisSim (5)	WARPED (7)
WITNESS (4)	XMLlab (4)		MS-MPI

[1] The use of citations to identify suitable programming environments is ill-defined and may be misleading, as a poor environment could be in principle cited frequently, and vice-versa.

In addition, even though it is not a specific tool for discrete event simulation, Microsoft Message Passing Interface (MS-MPI) was added to the list because it supports general parallel computing and is the preferred interface for Windows HPC Server 2008 R2.

The next step for evaluating programming environments was to collect the detailed information of each tool in the top 43 tools, based on a group of features related to the evaluation criteria described above, including name, last version and release date, description, parallel capability and approaches, interaction capability with MS/VB (Visual Basic), extendibility, languages of implementation, operating systems, GUI & visualization ability, application areas, customers, support documents, support web sites, license, developers and references. For more information refer to Annex B.

By analyzing the tools carefully and comparing to the evaluation criteria, a group of tools fell in the following categories and were rejected:

- Not compatible with Windows operating systems: e.g. ParaSol & WARPED;
- Used for specific application areas: e.g. GoldSim, NetSim, NS2/3, Plant Simulation, ProModel, Scilab, STELLA II & XMLlab;
- No parallel capability: e.g. AMESim, Arena, AutoMod, CSIM19/20, Dymola V, ExtendSim, Flexsim, GoldSim, GPSS/H, JiST, Plant Simulation, ProModel, Scicos, Scilab, SeSAm, SimCAD Pro, Simkit, SimScript III, Simula, SmallDEVS, STELLA II / iThink, Vensim, VisSim, WITNESS & XMLlab;
- Not updated for a long time or no continuous development: e.g. JiST, ParaSol & WARPED;
- No good accessible Web site found: e.g. C++Sim, Galatea, JDEVS, ModSim III, Simula & SmallDEVS;
- No good documentation found: e.g. Galatea & SeSAm;
- Under development: e.g. JDEVS.

Finally, the top 10 programming environments were identified as follows:

- AnyLogic,
- GridSim,
- MS-MPI,
- Parsec,
- Ptolemy II,
- SimEvents,
- Simio,
- Simjava,
- SimPy, and
- Simul8.

All the top 10 tools are Windows compatible, and implemented in common programming languages, such as, C, C++, C#, Visual Basic.NET, Java or Python. These popular tools are also application independent and provide the parallel capability of simulation execution with MPI, multi-threads or network sockets.

4.4.4 Recommending Programming Environments

To make the final recommendations of programming environments, further, rigorous evaluation and comparison were performed with the top 10 tools and the evaluation criteria. Four more tools

were further rejected including GridSim, Parsec, Ptolemy II and Simjava. Although they are also very good tools, compared to other tools in the top 10 list, they are either not recently updated, too research focused, or have too few applications or too few customers.

The final six tools are recommended and categorized as the following groups:

- Group I:
    - MS-MPI with visual studio, and
    - Simio.
- Group II
    - AnyLogic,
    - Simul8,
    - SimEvents, and
    - SimPy.

The Group I consists of MS-MPI with visual studio and Simio. To develop a flexible Tyche Parallelization Management Tool, MS-MPI is recommended first, because it is a powerful and standardized interface for parallel programming. Windows HPC Server 2008 R2 can support MS-MPI applications with various languages, e.g. VC++ and C#. A new Tyche Simulation Engine may be also developed with MS-MPI with Microsoft Visual Studio (C++ or C#) to partition all iterations in simulation instances into proper iteration groups that can be mapped onto Windows HPC Sever 2008 cluster computer nodes.

Simio is another tool with parallel capability and good compatibility with C# and .NET API. It can support parallel executions at multi-core level automatically. Users can extend it with .NET languages including VB or C#. Therefore, compared to other tools, the development efforts with Simio may be reduced because of such compatibility.

The Group II includes AnyLogic, Simul8, SimEvents and SimPy. AnyLogic is a Java-based integrated tool supporting multi-simulation runs on multi-cores in parallel. It provides three well-known modelling approaches including system dynamics, discrete event simulation and agent-based modeling. Simul8 is a powerful tool that can perform multiple runs on networked computers in parallel. SimEvents can also support multiple runs of simulation in parallel with MATLAB Parallel Computing Toolbox. SimPy is a Python-based open source simulation tools. It is simple and can support parallel simulation with parallel Python installed. Compared to other tool, it is simple and easy to maintain. However, Python is an interpreted language and it may be slower than C++ or Java.

In summary, the two groups of programming environments may be considered as alternative candidate tools for future Tyche development. If the compatibility with current Tyche GUI and Dashboard is the main emphasis, the first group of tools is strongly recommended. If a new future Tyche system with more potential and flexibility is the main focus, the tools in second group should be considered.

5 Further evaluation of functionality and speed of simulation development environments

5.1 Functionality review of simulation development environments

Based on the requirements of the Tyche system, DRDC CORA developed a functionality checklist for further evaluation of the recommended simulation development environments. The project team reviewed and analyzed the simulation development environments for the functionality in the checklist. This section describes the results of functionality review of the development environments including Visual Basic, Visual C++, Visual C#, AnyLogic, SimEvents, Simio, SimPy and Simul8.

The functionality checklist consists of various features associated with the operating environment, data handling, processing constructs, MS Excel support and random number generation in the development environments. Most of the functions mainly focus on the programming language level for simulation software development.

From the functionality level perspective, the simulation development environments can be classified into two groups, i.e. the low-level programming languages including Visual Basic, Visual C++, Visual C#, and Python used in SimPy, as well as the integrated high-level development environments comprising AnyLogic, SimEvents, Simio and Simul8.

The functionality of each low-level programming language was analyzed and compared directly to the items in the checklist. For most answers in the result checklists, some keywords, brief phrases or reference links were provided to explain the answers further.

Most of the high-level development environments are software systems with integrated graphical user interfaces. In most applications, users may generate their simulation models directly by clicking and dragging in the GUIs of the development environments rather than writing programming code. Thus, it is challenging, at their GUI level, to directly compare the items in the checklist and the functionality of the integrated development environments.

However, most of the integrated simulation development environments support some internal or common programming languages for model development, extension or customization, and at the programming language level, their functionality is comparable to the items in the checklist. In this case, the project team used an indirect method to evaluate the functionality of the development environments, i.e. comparing the functionality of the internal or common programming languages supported by the development environments with the functionality checklist. The results with this method do not directly relate to the functionality at the GUI level in the development environments, but they reflect the capability of the development environments at their programming level for model development or customization.

Table 4 summarizes the results of functionality comparison between the checklist and the simulation development environments. The detailed data for each development environment can be found in Appendix C.

Table 4: Results of functionality comparison between the Tyche checklist and the related simulation development environments

Category	Functionality	Sub-functions	Visual Basic	Visual C++	Visual C#	Any Logic (Java)	Sim Events (Matlab)	Simio (VC#)	SimPy (Python)	Simlu8 (Visual Logic)
Operating Environment	Native code compilation		yes	yes	yes	no	no	no	no	?
	Error handling		yes	yes	yes	yes	yes	yes	yes	?
	Symbolic debugging		yes	yes	yes	yes	yes	yes	yes	yes
	Native disk I/O support	Native file I/O	yes	yes	yes	yes	yes	yes	yes	yes
		Native folder I/O	yes	yes	yes	yes	yes	yes	yes	no
		Native drive I/O	yes	yes	yes	yes	yes	yes	yes	no
	Command line support		yes	yes	yes	yes	yes	yes	yes	?
	Spawned processing		yes	yes	yes	yes	yes	yes	yes	?
Data handling	Record definitions	Byte	yes	yes	yes	yes	no	yes	no	no
		Boolean	yes	yes	yes	yes	yes	yes	yes	no
		Integer	yes	yes	yes	yes	yes	yes	yes	yes
		Long	yes	yes	yes	yes	yes	yes	yes	yes
		Single	yes	yes	yes	yes	yes	yes	no	yes
		Double	yes	yes	yes	yes	yes	yes	yes	yes
		Date	yes	yes	yes	yes	yes	yes	yes	yes
		String	yes	yes	yes	yes	yes	yes	yes	yes
		Record definitions	yes	yes	yes	yes	yes	yes	yes	yes
		Complex data objects (classes)	yes	yes	yes	yes	yes	yes	yes	yes
	Dynamic collections	Adding/removing items dynamically	yes	yes	yes	yes	yes	yes	yes	yes
		"Count" property	yes	yes	yes	yes	no	yes	yes	yes
		Dynamic update of items	yes	yes	yes	yes	yes	yes	yes	yes
		Size limited only by available memory	no	no	no	no	no	no	no	?
	Variable scoping	global	yes	yes	yes	yes	yes	yes	yes	yes
		Modular	yes	yes	yes	yes	yes	yes	yes	?
		Procedural	yes	yes	yes	yes	yes	yes	yes	yes
Processing	Modules		yes	yes	yes	yes	yes	yes	yes	yes

Category	Functionality	Sub-functions	Visual Basic	Visual C++	Visual C#	Any Logic (Java)	Sim Events (Matlab)	Simio (VC#)	SimPy (Python)	Simlu8 (Visual Logic)
constructs	Procedures and functions	subroutines	yes	yes	yes	yes	yes	yes	yes	?
		functions	yes	yes	yes	yes	yes	yes	yes	yes
		By reference	yes	yes	yes	no	yes	yes	yes	?
		By value	yes	yes	yes	yes	yes	yes	yes	?
		User defined record structures	yes	yes	yes	yes	yes	yes	yes	yes
		Complex data objects	yes	yes	yes	yes	yes	yes	yes	yes
	Looping structures	FOR…NEXT	yes	yes	yes	yes	yes	yes	yes	yes
		DO WHILE	yes	yes	yes	yes	yes	yes	yes	yes
		REPEAT…UNTIL	yes	yes	yes	yes	no	yes	yes	yes
	Conditional constructs	IF…THEN…ELSE	yes	yes	yes	yes	yes	yes	yes	yes
		CASE statements	yes	yes	yes	yes	yes	yes	yes	no
		Complex logic structures	yes	yes	yes	yes	yes	yes	yes	yes
		Boolean functions within logical conditions	yes	yes	yes	yes	yes	yes	yes	yes
Miscellaneous support	MS Excel support		yes	yes	yes	yes	yes	yes	yes	yes
	Random number generation	Random numbers	yes	yes	yes	yes	yes	yes	yes	yes
		Adjustable seed value	yes	yes	yes	yes	yes	yes	yes	yes
Summary (yes)			41	41	41	39	37	40	38	28

From the functionality point of view, if the numbers of "yes" answers in the result checklists are sorted in descent order, the development environments can be sequenced as follows:

- Visual Basic[2], Visual C++ and Visual C#,
- Simio (VC#),
- AnyLogic (Java),
- SimPy (Python),
- SimEvents (Matlab), and
- Simul8 (Visual Logic).

5.2 Speed comparison of simulation development environments

To compare the speed of simulation development environments, similar to the functionality review in the previous section, they can be categorized into two groups, i.e. the low-level programming language group and the integrated high-level development environment group.

Regarding the speed comparison of the integrated high-level development environments including AnyLogic, SimEvents, Simio and Simul8, the project team searched the Internet for possible test cases or literature directly related to speed comparison. Unfortunately, no information was found.

Therefore, for the SimEvents and Simul8, they could not be evaluated within the scope of the analysis.

AnyLogic and Simio were implemented in Java and VC# respectively. They can also be extended and customized in the implementation languages. Therefore, the project team looked for the speed comparison literature directly associated with the programming languages used for the implementation of development environments, i.e. using Java for the indirect speed comparison of AnyLogic and VC# for the indirect speed comparison of Simio. This approach is not precise at the GUI level, but if multiple test cases can be found and summarized for the related programming languages, it can indirectly reflect the average speed of the development environments to a certain extent.

For the low-level programming language group, including Visual Basic, Visual C++, Visual C# and Python used in SimPy, the project team searched and found the literature of speed comparison.

There were 21 references found for the speed comparison of the related programming languages, each of which comprised a group of test cases used to compare speed, e.g. C, C++, VB, VC++, VC#, Java and Python. (Bruckschlegel, 2005; Burch, 2004; Bytes, 2011; Cherrystone Software, 2010; Corlan, 2011; Cowell-Shah, 2004; Cplusplus, 2002; Debian, 2011; Fourment & Gillings,

[2] Note that for future programming, we refer to VB.NET, not the unsupported VB6 that Tyche is written in currently.

2008; Github, 2011; Javaworld, 2011; Lewis & Neumann, 2003; Morin, 2011; Paquet, 2011; Sestoft, 2010; Tommti-systems, 2011; Twistedmatrix, 2011; VBForums, 2002; Volkerschatz, 2011; Wilkinson, 2011; Xan, 2007.)

The test cases in the retrieved literature consisted of various data structures, operations and algorithms, and were executed on various operating systems and hardware platforms with different versions of compilers/interpreters of the programming languages. The main features in the retrieved test cases are as follows:

- Operations tested: integer arithmetic, double arithmetic, long arithmetic, trigonometric functions, large integer arithmetic, file I/O, arrays, exceptions, hash maps, sorting, list, polynomial calculations, matrix multiplications, logic operations, conditional statements, various loops including nested loops, string concatenation, dynamic memory allocation, object creation, destruction, method call, standard I/O, database access, accessing system resources, etc.;
- Platforms used for testing: Windows, Linux, Mac OS, etc.;
- Versions of programming languages used for testing: several recent versions; and
- Hardware used for testing: various computer systems, e.g. Intel, AMD, 32-bit, 64-bit, etc.

The simulation tools evaluated in this project were implemented in various programming languages. Most simulation tools have been evolving for many years. As a result, they might use various versions of the related programming languages. If only the most recent version of programming languages were used to compare their speed, there might be a larger gap between the result of comparison and the actual speed of the simulation environment. Therefore, this project attempted to compare the speed of popular programming language used in the simulation environment at a high level, i.e. the average speed of the programming languages based on the speed comparison literature in last 10 years to match the evolving time of the simulation tools and reduce the difference between the speed comparison results of programming languages and the actual speed of the simulation tools. Many detailed comparisons are also possible, such as the comparison of most recent versions of all programming languages, the comparison on similar hardware platforms, the comparison of selected programming language pairs, and the comparison for a group of specific operations. They could be further evaluated in a future project.

The literature-based comparison of programming languages is a challenging task because:

- there is no standard for the literature-based speed comparison of programming languages;
- there is no recognized organization that evaluates and publishes the comparison results of programming languages;
- each popular language always keeps evolving; and

- the execution time of test cases varies widely depending on many factors, e.g. operating systems, hardware, versions of programming languages, programmer's programming skills, and operations to test.

This project proposed an approach to handle the complexity. The basic idea for speed evaluation was to develop an approach to convert the inconsistent execution time of the programming languages in various test cases in the literature into a unified quantitative measurement, and to summarize and rank the unified quantitative measurement for speed comparison. Before describing the approach to evaluate the speed of the related programming languages, several terms are defined below.

Definition 1 - *Test case in literature*: A *test case in literature* or simply a *test case* consists of a group of computer programs implemented in different programming languages for a specific operation or algorithm to compare the speed of the programming languages. For example, a test case may include a sort algorithm implemented in C, Java and Python to compare the speed of the three languages for the algorithm.

Definition $_{\text{two}}$ – *Literature-based experiment:* A *Literature-based experiment*, or simply an *experiment*, refers to one or more test cases and the associated testing results reported in a literature for comparing the speed of a group of programming languages for a set of operations or algorithms. In other words, each retrieved literature associated with the speed comparison of programming languages is considered as an experiment.

Definition $_{\text{three}}$ – *Relative execution time of a programming language, denoted as R*: The *relative execution time* of a programming language in an experiment or a test case refers to the ratio between the execution time taken by the programming language and the execution time of the fastest programming language in the same experiment or test case, i.e.,

$$R = ExecutionTimeOfALanguageInAnExperiment \,/\, FastestExecutionTimeInTheExperiment \quad (1)$$

Obviously, the minimum value of R is 1.0, and the larger the R is, the slower the language is.

Definition $_{\text{four}}$ – *Relative speed factor, denoted as F*: A *relative speed factor,* or simply a *speed factor,* is a measurement of relative speed of a programming language in an experiment. The *relative speed factor* is simply defined as the inverse number of *relative execution time*, i.e.,

$$F = 1 / R. \quad (2)$$

Apparently, the minimum value of F is 0.0 and the maximum value of F is 1.0. The larger the *relative speed factor* of a programming language in an experiment, the faster the programming language.

Definition $_{\text{five}}$ – *Speed of a programming language to another programming language in a group of experiments*: The *speed of a programming language, Lx, to another programming language, Ly, in a group of experiments* denoted as S_{LxtoLy}, is a measurement to represent the degree that the programming language, *Lx*, is faster than the programming language, *Ly*, in the group of experiments. If *Lx* is faster than *Ly* on average in the group of experiments, S_{LxtoLy} is computed with the average differences of *relative speed factor*s between the language *Lx* and the

languages *Ly* in all the experiments. If *Lx* is slower than *Ly* on average in the group of experiments, S_{LxtoLy} is defined as 0.

For example, there are two experiments for the speed comparison of C and Java. The *relative speed factor* of C is 1.0, and the *relative speed factor* of Java is 0.59 in the first experiment. The *relative speed factor* of C is 0.98, and the *relative speed factor* of Java is 0.66 in the second experiment. The *speed of C to Java* in the two experiments is computed as follows,

$$S_{CtoJava} = ((1.0 - 0.59) + (0.98 - 0.66)) / _{\text{two}} = 0.365.$$

Definition $_{\text{six}}$ **–** ***Overall speed of a programming language to all other languages in a group of experiments*****:** The *overall speed (*or simply *speed) of a programming language, Lx, to all other languages in a group of experiments*, denoted as S_{Lx} or *S*, is a measurement to represent the degree that the programming language, *Lx*, is faster than all the other programming languages in the group of experiments. If the programming language, *Lx*, is the slowest in the group of experiments, the *overall speed* of *Lx* is 0. If there are one or more programming languages that are slower than the programming language, *Lx*, in the group of experiments, the *overall speed* of the programming language, S_{Lx}, is the sum of the *speed of the programming language* to each other programming language in the group of experiments. In the coming paragraphs, some examples for the computing of *overall speed* of a programming language will be given for further explanation.

The minimum value of *overall speed* is 0.0. The upper boundary of *overall speed* in a language group with *N* languages is (*N - 1*).

This approach for the speed comparison of programming languages uses the relativity of execution time in a group of test cases to eliminate the differences of operating systems, hardware platforms, operations to test, and programmers' skills. Therefore, the comparison results with this approach reflect the average speed of the programming languages to a great extent.

The following list depicts the method and steps to evaluate the speed of the related programming languages:

- Review literature and extract the information that directly relates to the speed comparison of the programming languages associated with the selected simulation development environments;
- Identify all test cases in the literature/experiment;
- Recognize the results of execution time of test cases in the experiment;
- For each test case in the experiment, if the test case for a programming language is executed multiple times on different platforms or using different compilers/interpreters, compute the average execution time of test cases for the programming language in the experiment;
- Combine the average execution time of all test cases for each programming language in the experiment to get the overall average execution time of the programming language in the experiment;
- Compute the *relative execution time, R*, for each programming language in the experiment;
- Calculate the *relative speed factor*, *F*, for each programming language in the experiment;

- Summarize all the *relative speed factors* for each programming language in all experiments;
- Compute the *speed* of each programming language, *Lx*, to another programming language, *Ly*, in all experiments, i.e. S_{LxtoLy};
- Compute the *overall speed*, i.e. *S*, for each programming language based on its *speed* to other languages in all experiments;
- Rank the final *overall speed* for all the related programming languages.

The following example explains this method in detail. For example, the literature from Bruckschlegel (2005) developed 16 test cases for various operations in C, VC++, VC# and Java to compare their speed. The following Figure 23 and Figure 24 are two groups of average execution time in the literature for non-arithmetic and arithmetic operations respectively.

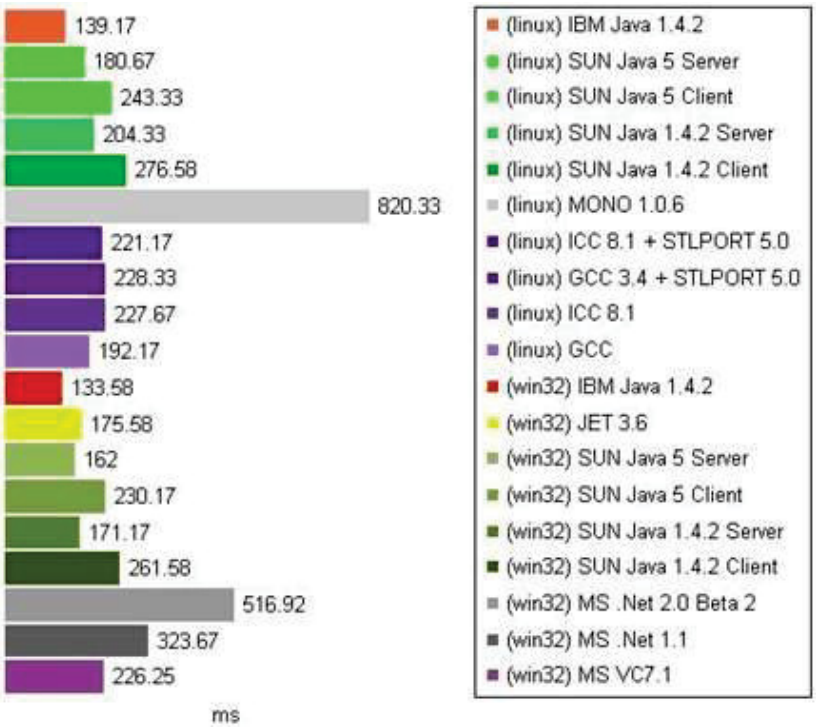

Figure 23: Group A; Average execution time (in milliseconds) of the test cases without arithmetic operations

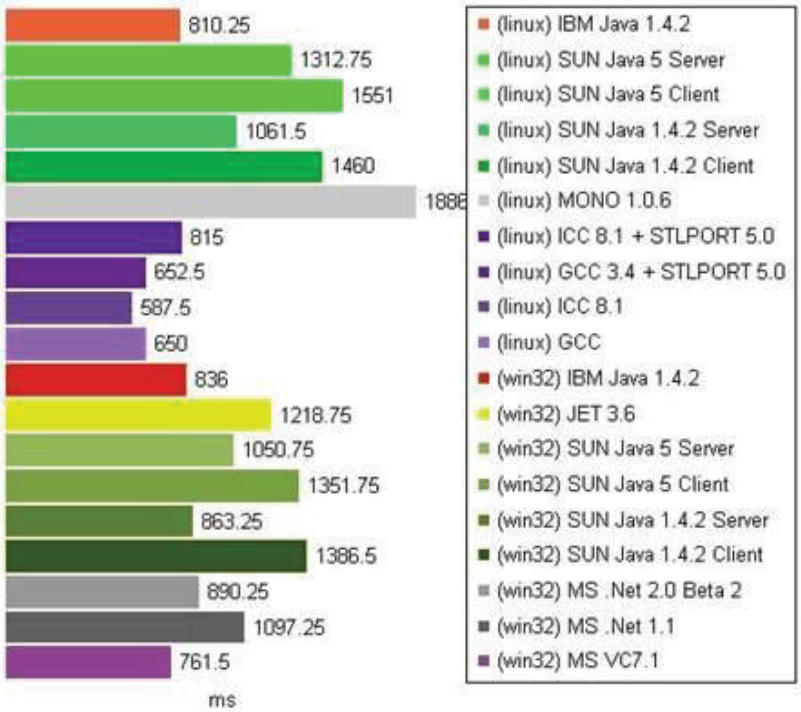

Figure 24: Group B: Average execution time of the test cases for arithmetic and trigonometric operations.

In the Group A, for example, there were four testing results for the C programming language with different compilers and platforms, including the (linux) ICC8.1+STLPORT5.0, (linux) GCC3.4+STLPORT5.0, (linux) ICC8.1 and (linux) GCC. In our approach, the average execution time of C in this group was computed based on the four testing results, i.e.

$$AvgTimeOfC = (221.17+228.33+227.67+192.17)/4 = 217.34 \text{ ms.}$$

We used the same method to compute the average execution time of other programming languages in the group. The result was 198.02 ms in Java, 226.25 ms in VC++ and 553.64 ms in VC# respectively, as shown in Table 5.

The overall average execution time of each programming language in the experiment was computed based on the average execution time of the programming language in the two groups, and the results were 446.80 in C, 493.88 in VC++, 922.74 in VC# and 685.49 in Java respectively, as shown in Table 5. The *relative execution time* of each language was the division of the overall average execution time of the language by the average execution time of the fastest language, i.e. C, in the group. Finally, the *relative execution time* and the *relative speed factors* of the four languages in the experiment are obtained as shown in Table 5 . The *relative execution time* was 1.0 for C, 1.1054 for VC++, 2.0652 for VC#, and 1.5342 for Java respectively. The final *relative speed factor*s in the experiment were 1.0 for C, 0.9046 for VC++, 0.4842 for VC# and 0.6518 for Java respectively.

Table 5: Average execution time, relative execution time and relative speed factors of programming languages in the Bruckschlegel experiment

	C	VC++	C#	Java
Average execution time for Group A	217.34	226.25	553.64	198.02
Average execution time for Group B	676.25	761.5	1291.83	1172.95
Overall average execution time	446.80	493.88	922.74	685.49
Relative execution time, *R*	1.0	1.1054	2.0652	1.5342
Relative speed factor, *F*	1.0	0.9046	0.4842	0.6518

We used the same method for all the test cases in the retrieved literature, and obtained the results of *relative speed factors* for all the related programming languages as shown in Table 6.

Table 6: Summary of the relative speed factors of the related programming languages

	VC++ (*F*)	C (*F*)	C++ (*F*)	VB (*F*)	Java (*F*)	VC# (*F*)	Python (*F*)
Bytes2011	1.0	.7309		.5582	.5173	.7351	0.0317
Cowell-Shah2004	1.0	.6685		.5681	.4733	.7473	0.0933
Debian2011		1.0	0.9192		0.6413	0.4143	0.0220
Paquet2011		1.0	0.8633		0.1300	0.1926	0.0225
Fourment Gillings 2008		1.0	0.5684		0.0640	0.2324	0.0129
Github 2011		1.0			0.3865	0.0605	0.0349
Cherry stone 2010		1.0	0.374		0.2776	0.2343	
Bruck schelgel 2005	0.9046	1.0			0.6518	0.4842	
Wilkinson 2011		1.0			0.6982		0.0347
Corlan 2011		1.0			0.5631		0.0094
Burch 2004		1.0			0.5076		0.0222
Tommti Systems			1.0		0.5146	0.7173	

	VC++ (*F*)	C (*F*)	C++ (*F*)	VB (*F*)	Java (*F*)	VC# (*F*)	Python (*F*)
2011							
Sestoft 2010		1.0			0.7458	0.4977	
Volker schatz 2011		1.0			0.2226		
Lewis Neumann 2003		1.0			0.5463		
Cplusplus 2002			1.0		0.7336		
Xan 2007			1.0		0.5481		
Javaworld 2011			0.7785		1.0		
VBForuns 2002				0.7717		1.0	
Morin 2011	1.0					0.4552	
Twisted matrix 2011					1.0		0.7621

The computation of the *overall speed* of a programming language compared to all other languages in a group of experiments is arrived at by the following summarized process.

- For each programming language, *Lx*, involved in the experiments, identify all the programming language that are slower than the programming language *Lx* in each experiment, denoted as Ly_1, Ly_2, ... For example, if the Visual Basic is taken into account, the Java and Python are slower than Visual Basic in the Bytes 2011 experiment; and the Java and Python are also slower than VB in the Cowell-Shah 2004 experiment, as shown in Table 6;
- Compute the difference of *relative speed factors* between the programming language *Lx* and each of the slower language Ly_i in each experiment; For example, the difference of *relative speed factors* between VB and Java in the Bytes 2011 experiment is 0.5582-0.5173 = 0.0409; and the difference of *relative speed factors* between VB and Java in the Cowell-Shah 2004 experiment is 0.5681 – 0.4733 = 0.0948;
- Calculate the *speed* of the programming language *Lx* compared to each of the slower language Ly_i, denoted as $S_{LxtoLyi}$, in all experiments. $S_{LxtoLyi}$ is the average difference of *relative speed factors* between the language *Lx* and the slower language Ly_i in all experiments, i.e.

  $$S_{LxtoLyi} = SumOfTheDifferencesOfSpeedFactorsBetweenLxAndLy_iInEachExperiment \,/\, NumberOfRelatedExperiments;$$

  For example, the *speed of VB to Java* in all the two related experiments is

  $$S_{VBtoJava} = (0.0409 + 0.0948) / _{two} = 0.0679;$$

- Compute the *overall speed* of the programming language *Lx* compared to all the slower languages than *Lx* in all experiments. The *overall speed* of the programming *Lx* to all the

slower languages is the sum of the *speed* of the programming language to each of the slower language in all experiments, i.e.

- $$S_{LxtoAllLanguages} \text{ or } S_{Lx} = S_{LxtoLy1} + S_{LxtoLy2} + S_{LxtoLy3} +...$$

  where, Ly_1, Ly_2, Ly_3, ... are the slower languages than Lx.

As a complete example, let us summarize the whole computation process for the *overall speed* of Visual Basic to explain the method further. The *relative speed factor*s of Visual Basic in Table 6 are extracted and re-displayed in Table 7.

Table 7: Relative speed factors compared to that of Visual Basic

	VB	Java	VC#	Python
Bytes, 2011	.5582	.5173	.7351	.0317
Cowell-Shah, 2004	.5681	.4733	.7473	.0933
VBForums 2002	0.7717		1.0	

There were three experiments for the speed comparison between Visual Basic, Java, VC# and Python, including Bytes, 2011, Cowell-Shah, 2004 and VBForums, 2002. In the three experiments, VB was faster than Java & Python in the Bytes, 2011, and it was also faster than Java & Python in the Cowell-Shah, 2004. However, VB was not faster than VC# in the three experiments. Therefore, we got the *speed* of VB to Java,

$$S_{VBtoJava} = ((0.5582 - 0.5173) + (0.5681 – 0.4733))/2 = 0.0679,$$

and the *speed* of VB to Python,

$$S_{VBtoPython} = ((0.5582 – 0.0317) + (0.5681 – 0.0933)) / _{\text{two}} = 0.5007.$$

Finally, the *overall speed* of VB to all the slower languages than VB was

$$S_{VB} = S_{VBtoJava} + S_{VBtoPython} = 0.5686,$$

as shown in Table 8.

Table 8: Computation process of overall speed of Visual Basic

	Java	VC#	Python	*Overall Speed* (*S*) of VB
VB	2-0 $S_{VBtoJava}$: 0.0679		2-0 $S_{VBtoPython}$: 0.*5007*	0.5686

The "2-0" in the VB-Java cell in Table 8 means that there were two experiments in which VB was faster than Java and there was no experiment in which VB was slower than Java.

Table 9 summaries the computation process of *overall speed* for all the related programming languages.

Table 9: Speed and overall speed of programming languages

	C	C++	VB	Java	VC#	Python	*Overall Speed* (*S*)
VC++	2-1 $S_{VC++toC}$: 0.1700		2-0 $S_{VC++toVB}$: 0.4368	3-0 $S_{VC++toJava}$: 0.4525	3-0 $S_{VC++toVC\#}$: 0.3445	2-0 $S_{VC++toPython}$: 0.9375	2.1714
C		4-0 S_{CtoC++}: 0.3189	2-0 S_{CtoVB}: 0.1367	14-0 $S_{CtoJava}$: 0.4981	7-2 $S_{CtoVC\#}$: 0.5332	9-0 $S_{CtoPython}$: 0.9027	2.3896
C++				7-1 $S_{C++toJava}$: 0.1979	5-0 $S_{C++toVC\#}$: 0.3867	3-0 $S_{C++toPython}$: 0.8745	1.4591
VB				2-0 $S_{VBtoJava}$: 0.0679		2-0 $S_{VBtoPython}$: 0.5007	0.5686
Java					5-5 $S_{JavatoVC\#}$: 0.0087	10-0 $S_{JavatoPython}$: 0.3936	0.4023
VC#			3-0 $S_{VC\#toVB}$: 0.1948			6-0 $S_{VC\#toPython}$: 0.3658	0.5606
Python							0

The final results of the speed comparison for the six programming languages related to the selected simulation development environments are shown in Table 10, and the corresponding chart of *overall speed* is shown in Figure 25.

Table 10: Overall speed of the six related programming languages

	VC++	VB	VC#	VC#, (Simio)	Java, (AnyLogic)	Python (SimPy)
Overall Speed	2.1714	0.5686	0.5606	0.5606	0.4023	0.0
Normalized *Overall Speed*	1.0	0.2619	0.2582	0.2582	0.2215	0.0

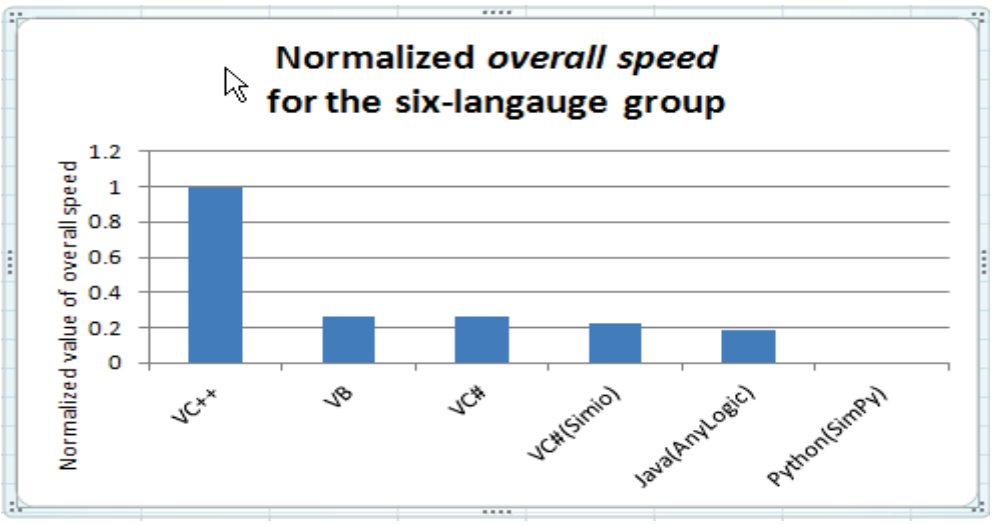

Figure 25: Results of speed comparison between the six related programming languages

Obviously, the ranking of *overall speeds* of the six related programming languages is as follows:

- VC++,
- VB ,
- VC#, VC# (Simio),
- Java (AnyLogic), and
- Python (SimPy).

The results in Figure 25 reflect the *overall speed* of each programming language within the six-language group, rather than one-to-one comparison. If a different combination of languages is chosen, there may be some differences between results. For example, if we choose only the VB, VC++ and VC# as a group, as shown in Table 11 and Figure 26, the results of VB and VC# are different between the six-language group and the three-language group. The *overall speed* of VB is a little bit bigger than VC# in the six-language group. However, in the three-language group, VC# is faster than VB. The main reason for this difference is that, in the six-language group, VB was relatively faster than VC# in the experiments comparing them with Python.

Table 11: Result speed for the VB, VC++ and VC# group

	VB	VC#	*Overall Speed (S)*	*Normalized Overall Speed*
VC++	2-0 $S_{VC++toVB:}$ 0.4368	3-0 $S_{VC++toVC\#}$: 0.3445	0.7813	1.0
VB			0	0
VC#	3-0 $S_{VC\#toVB}$ 0.1948		0.1948	0.2493

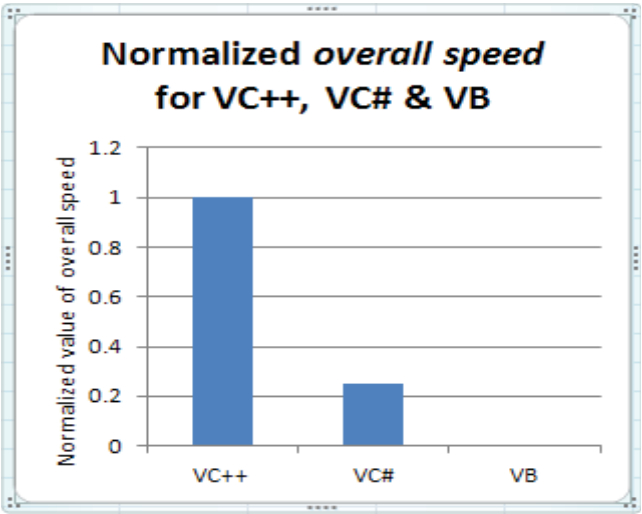

Figure 26: Results of speed comparison between VB, VC++ and VC#

In summary, no direct test case for speed comparison was found for AnyLogic, SimEvents, Simio and Simul8. Therefore, they could not be evaluated within the scope of the analysis. However, AnyLogic and Simio are implemented in Java and VC# respectively, and they can also be extended and customized with the related programming languages. Thus, Java and VC# were used for indirect speed comparison of AnyLogic and Simio with other development environments.

For all the related low-level programming languages in the selected simulation development environments, including VB, VC++, VC#, Java and Python, 21 references for speed comparison were found, each of which used a group of test cases to compare the speed of a sub-set of the related programming languages. The results of speed comparison show that the VC++ is the fastest programming language in the selected language group, as shown in Figure 25 and Figure 26. VC# is a little bit faster than VB in the retrieved experiments. Java is slower than VB, and Python is the slowest language in the selected language group because it is an interpretation-based language.

6 Summary and Next Steps

This project first analyzed the current Tyche simulation system and identified the problems that influence simulation execution time. Three categories of recommendations were proposed to speed up the Tyche simulation and develop the future Tyche system easily and quickly.

The results of Tyche system analysis were summarized as an entity/conceptual model, a dynamic component model and a detailed simulation process.

Two types of problems were identified based on the Tyche analysis: one was the problem of single-thread execution, and another was the issues of redundant or repeated operations within iterations.

This project proposed two kinds of approaches to solve the problems: a parallel architecture to handle the single-thread problem, and a group of optimized functions for simulation engine to improve the performance of iterations. This newly proposed architecture divides all iterations in simulation instances into parallel groups to makes full use of the parallel capabilities of Windows HPC Server 2008 R2 to speed up simulation execution. Specifically, three partitioning options for the simulation instances are reviewed. The optimized functions for simulation engine reduce redundant and repeated operations from iteration to iteration. Three alternative methods are suggested for handling output data within iterations quickly, including reducing times of file access, encoding output text and multi-threading. To make the future Tyche development easy and quick, this project evaluated and recommended a group of programming environments. Based on Internet searching, 22 survey/review/summary papers or documents about libraries, languages, platforms or software for DES/PDES were collected, which covered 350 tools. With the criteria defined by DRDC CORA and the project team, top 43 popular tools were evaluated in detail. Finally, the top six tools were identified and categorized as two groups of recommended programming environments, including MS-MPI and Simio as the first group, and AnyLogic, Simul8, SimEvents & SimPy as the second group. If the compatibility with current Tyche GUI and Dashboard is the main emphasis, the first group of tools is strongly recommended. If a new future Tyche system with more potential and flexibility is the main focus, the tools in second group should be considered.

Based on the recommended simulation development environments, additional tasks were conducted by the project team for a further evaluation of the functionality (as a proxy for portability) and literature-based speed comparison among the simulation development environments. Compared to the functionality checklist provided by Scientific Authority, Visual Basic, Visual C++ and Visual C# provide more functions in the checklist. For the literature-based speed comparison of simulation development environments, no direct test case for speed comparison was found for AnyLogic, SimEvents, Simio and Simul8. Therefore, they could not be evaluated within the scope of the analysis. However, the project team evaluated all the low-level programming languages related to the simulation development environments. The results show that VC++ is the fastest programming language compared to VB, VC#, Java and Python. VB and VC# are quite similar, and Java is a little bit slower than VB & VC#. The Python is the slowest programming language because it is an interpreted language.

Based on the Tyche system analysis and recommendations, the following tasks are identified as next steps for developing a fast and powerful Tyche simulation system in future projects:

- Implement the proposed parallel architecture with Windows HPC Server 2008 R2 and one of the recommended programming environments;

- Implement a prototype parallel and optimal simulation engine with the suggested approaches; and

- Test and evaluate the integrated high-level simulation development environments, including AnyLogic, SimEvents, Simio and Simul8, for a detailed comparison of the speed and adaptability of future Tyche development.

References

Allen, D., Eisler, C. & Forget, A. (2006), *A Users Guide to Tyche Version 2.0.* DRDC CORA Report, TR 2006-14.

Altmann, M. (2011). List of software for computer simulations. Retrieved on July 12, 2011, from http://oldwww.rasip.fer.hr/nastava/mis/comp-simulation-software-FAQ.html.

Ammar, H. & Deng, S. (1992). Time Warp simulation using time scale decomposition. *ACM Transactions on Modeling and Computer Simulation*, 2(2):158-177.

Bagrodia, R. Chandy, K.M. & Liao, W.T. (1991). A unifying framework for distributed simulation. *ACM Transactions on Modeling and Computer Simulation,* 1(4):348-385.

Banks, J., Carson, J., Nelson, B. & Nicol, D. (2009), *Discrete-Event System Simulation,* 5th edition. Prentice Hall.

Barney, B. (2010). *Introduction to Parallel Computing.* Retrieved from https://computing.llnl.gov/tutorials/parallel_comp/, on June 14, 2011.

Barr, R., Haas, Z.J. & van Renesse, R. (2004). JiST: An efficient approach to simulation using virtual machines. *Software – Practice and Experience,* 00:1-7.

Bruckschlegel, T. (2005). Microbenchmarking C++, C#, and Java. Retrieved from http://drdobbs.com/184401976, on Sep. 28, 2011.

Burch, C. (2004). Evaluating language efficiency. Retrieved from http://ozark.hendrix.edu/~burch/cs/360/assn/proj2-soln.pdf, on Sep. 29, 2011.

Buyya, R. & Murshed, M. (2002). GridSim: A toolkit for the modeling and simulation of distributed resources management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14:1175-1220. Retrieved from http://www.buyya.com/papers/gridsim.pdf, on August 3, 2011.

Bytes. (2011). Not exactly the same performance. Retrieved from http://bytes.com/topic/c-sharp/answers/236257-c-vb-net-not-exaclty-same-performance,

CACI. (2011a). SIMPSCRIPT III. Retrieved from http://www.simscript.net/products/products.html, on August 3, 2011.

CACI. (2011b). SIMPSCRIPT III Product Description. Retrieved from http://www.simscript.net/products/simscript_description.html, on August 3, 2011.

CACI. (2011c). SIMSCRIPT III: New Modular Object-Oriented Simulation Language. Retrieved from http://www.simscript.com/docs/SIMSCRIPT_III_Object%20Oriented.pdf, on August 3, 2011.

Cambridge SRG. (2011). Parallel Discrete Event Simulation. Cambridge University, Computer Laboratory, Systems Research Group - NetOS. Retrieved on July 11, 2011, from http://www.cl.cam.ac.uk/research/srg/netos/plana/PDES.html.

Chandy, K.M. & Sherman, R. (1989). Space-time and simulation. *Proceedings of the 1989 SCS Multiconference on Distributed Simulation,* 21(2):53-57.

Cherrystone Software. (2010). Algorithm performance comparison between C, C++, Java and C# programming languages. Retrieved from http://www.cherrystonesoftware.com/doc/AlgorithmicPerformance.pdf, on Sep. 28, 2011.

Comfort, J.C. (1984). The simulation of a master-slave event set processor. *Simulation*, 42(3):117-124.

Constitution. (2011). Simulation tools for Linux systems. Retrieved on July 11, 2011, from http://www.constitution.org/comp/simtools.htm.

Corlan. (2011). Programming language benchmarks. Retrieved from http://dan.corlan.net/bench.html, on Sep. 29, 2011.

Cowell-Shah, C.W. (2004). Nine language performance round-up: benchmarking math & file I/O. Retrieved from http://www.osnews.com/story/5602, on Sep. 30, 2011.

Cplusplus. (2002). When to use what language and why. Retrieved from http://www.cplusplus.com/articles/42E1wA7f/, on Sep. 28, 2011.

Dave, J. (2005). *Parallel Discrete Event Simulation Techniques for Scientific Simulations.* Thesis of Master of Science in Computer Science of Georgia Institute of Technology. Georgia, USA. Retrieved from http://etd.gatech.edu/theses/available/etd-04152005-165207/unrestricted/dave_jagrut_d_200505_mast.pdf, on June 13, 2011.

Debian. (2011). Computer language benchmarks game. Retrieved from http://shootout.alioth.debian.org/, on Sep. 30, 2011.

Dudenhoeffer, D.D., Permann, M.R., & Sussman, E.M. (2002). A parallel simulation framework for infrastructure modeling and analysis*, Proceedings of the 2002 Winter Simulation Conference (WSC'02)*, Vol.2:1971-1977, December, San Diego, CA, USA.

Ferscha, A. (1995). Parallel and distributed simulation of discrete event systems. In *Handbook of Parallel and Distributed Computing.* McGraw-Hill, 1995.

Filippi, J.-B., Bernardi, F. & Delhom, M. (2011). The JDEVS modeling and simulation environment. Retrieved from http://www.iemss.org/iemss2002/proceedings/pdf/volume%20tre/17_filippi.pdf, on August 3, 2011.

Foster, I. (1995), Designing and Building Parallel Programs – Concepts and Tools for Parallel Software Engineering. Addison Wesley.

Fourment, M. And Gillings, M.R. (2008). A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics,* 9:82. Retrieved from http://www.biomedcentral.com/1471-2105/9/82, on Sep. 28, 2011.

Fujimoto, R.M. (1990). Parallel discrete event simulation. *Communications of the ACM,* 33(10):30-53.

Fujimoto, R.M. (1993). Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3):213-230.

Fujimoto, R. M., (2000), *Parallel and Distributed Simulation Systems.* John Wiley & Sons, Inc., New York.

Github. (2011). Programming languages benchmarks. Retrieved from http://attractivechaos.github.com/plb/, on Sep. 28, 2011.

Goble, J. (1997). MODSIM III – A TUTORIAL. *Proceedings of the 1997 Winter Simulation Conference*, 601-605. Retrieved from http://www.informs-sim.org/wsc97papers/0601.PDF, on August 3, 2011.

Goes, L.F.W., Pousa, C.V., Carvalho, M.B., Ramos, L.E.S., & Martins, C.A.P.S. (2011a). JSDESLib: A Library for the Development of Discrete-Event Simulation Tools of Parallel Systems. *Proceedings, of the 19th IEEE International Parallel and Distributed Processing Symposium*. Retrieved on July 12, 2011, from http://www.research.rutgers.edu/~luramos/pdf/ipdps05jsdeslib.pdf

Goes, L.F.W., Ramos, L.E.S. & Martins, C.A.P.S. (2011b). ClusterSim: A Java-Based Parallel Discrete-Event Simulation Tool from Cluster Computing. Proceedings of the 2004 IEEE International Conference on Cluster Computing. Retrieved on July 12, 2011, from http://www.research.rutgers.edu/~luramos/pdf/iccc04clustersim.pdf

Gomes, F. (2011). Simulation bookmarks. Department of Computer Science, University of Calgary. Retrieved on July 12, 2011, from http://pages.cpsc.ucalgary.ca/~gomes/HTML/sim.html

Heidelberger, P. & Stone, H.S. (1990). Parallel trace-driven cache simulation by time partitioning. *Proceedings of the 1990 Winter Simulation Conference (WSC'90),* 734-737. December, New Orleans, LA, USA.

Henderson, S.G. (2003). Input model uncertainty: why do we care and what should we do about it? *Proceedings of the 2003 Winter Simulation Conference (WSC'03),* Vol.1:90-100, December, New Orleans, LA, USA. Retrieved from http://docs.google.com/viewer?a=v&q=cache:uFqJVxEBBUoJ:citeseerx.ist.psu.edu/viewdoc/download%3Fdoi%3D10.1.1.130.947%26rep%3Drep1%26type%3Dpdf+input+model+uncertainty:+why+do+we+care+and+what+should&hl=en&gl=ca&pid=bl&srcid=ADGEESjsmHaUWf-4iJKU45bOpinNvZCzXi9_ql8APQUr9YxdnK_n47XyKWiIgi2Fo6ehUNsc4RLL_QQqhbwLxuO8JZ-zCkkFBbfB0A8nrVlnduXR8t9-ga3vkmcX5dR9v23Jp4FANJXU&sig=AHIEtbTd9Soy9uclDZfWolEgC0GO7cHfAg, on June 13, 2011.

Heppenstall, D. (2007), A User's Guide and Programmer's Manual to Tyche Version 2.2. DRDC CORA Report, CR 2007-01.

Horejsi, P. (2011). Software Architecture for Parallel Simulation Optimization. Retrieved from http://www.mmscience.eu/archives/MM_Science_201009.pdf, on July 27, 2011.

Hybinette, M. & Fujimoto, R. (2001). Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378:407.

Isee systems. (2011a). STELLA. Retrieved from http://www.iseesystems.com/softwares/Education/StellaSoftware.aspx, on August 4, 2011.

Isee systems. (2011b). System Thinking and the STELLA Software. Retrieved from http://www.iseesystems.com/resources/Articles/STELLA%20IST%20-%20Chapter%201.pdf, on August 3, 2011.

Isee systems. (2011c). iThink – Systems Thinking for Business. Retrieved from http://www.iseesystems.com/softwares/Business/IthinkSoftware.aspx, on August 3, 2011.

Jain, R. (1991). The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. John Wiley & Sons.

Janousek, V. & Kironsky, E. (2006). Exploratory Modeling with SmallDEVS. Retrieved from http://www.fit.vutbr.cz/~janousek/publications/2006-esm-smalldevs.pdf, on August 3, 2011.

Javaworld. (2011). Performance tests show Java as fast as C++. Retrieved from http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf.html, on Sep. 30, 2011.

Kiesling, T. & Luthi, J. (2005). Towards time-parallel road traffic simulation. *Proceedings of the 19th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2005),* 7-15, June, Monterey, CA, USA.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM,* 21(7):558-565.

Lanner. (2011). Decision analysis with WITNESS simulation: product development process modeling. Retrieved from http://www2.humusoft.cz/download/white-papers/witness/decision_anlysis_with_witness.pdf, on August 3, 2011.

Lantz, E. (2008). Windows HPC Server 2008 – Using Microsoft Message Passing Interface (MS-MPI). Retrieved from http://www.microsoft.com.br/kitservidores/infra/HPC%20Pack%202008%20R2%20Enterprise/Windows%20HPC%20Server%202008%20-%20Using%20MS%20MPI%20White%20Paper(Final).doc, on August 3, 2011.

Law, A.M. & Kelton, W.D., (1991), *Simulation Modeling and Analysis.* McGraw-Hill, second edition, 1991.

Lee, E.A. & Neuendorffer, S. (2007). Tutorial: Building Ptolemy II Models Graphically. Technical Report UCB/EECS-2007-129, EECS, University of California at Berkeley. Retrieved from http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-129.pdf, on August 3, 2011.

Lewis, J.P. and Neumann, U. (2003). Performance of Java versus C++. Retrieved from http://scribblethink.org/Computer/javaCbenchmark.html, on Oct. 3, 2011.

Lin, Y.B. & Fishwick, P. (1996). Asynchronous parallel discrete event simulation. *IEEE Transactions on Systems, Man, and Cybernetics,* 26(4):397-412.

Lin, Y.B. & Lazowska, E.D. (1991). A time-division algorithm for parallel simulation. *ACM Transactions on Modeling and Computer Simulation,* 1(1):73-83.

Liu, J. (2010). Parallel Discrete-Event Simulation. *Wiley Encyclopedia of Operations Research and Management Science (book chapter).* Retrieved from http://users.cis.fiu.edu/~liux/research/papers/pdes-eorms09.pdf, on June 13, 2010.

Low, Y.H., Lim, C.C., Cai, W. Huang, Y., Hsu, W.J., Jain, S. & Turner, S.J. (1999). Survey of languages and runtime libraries for parallel discrete-event simulation. *Simulation and Transactions of the Society for Computer Simulation (SCS), Joint Special Issue on Parallel and Distributed Simulation*, 72(3):170-186.

Malik, A.W., Park, A.J. & Fujimoto, R.M. (2010). An optimistic parallel simulation protocol for cloud computing environments. *SCS M&S Magazine*, 4:1-9. Retrieved from http://www.scs.org/magazines/2010-10/index_file/Files/Fujimoto.pdf, on June 10, 2011.

Markt, P.L. & Mayer, M.H. (1997). WITNESS simulation software – a flexible suite of simulation tools. *Proceedings of the 1997 Winter Simulation Conferences*, 711-717. Retrieved from http://www.informs-sim.org/wsc97papers/0711.PDF, on August 3, 2011.

MathWorks. (2011a). Model and simulation discrete-event systems. Retrieved from http://www.mathworks.com/products/simevents/, on July 27, 2011.

MathWorks. (2011b). Parallel Simulation. Retrieved from http://www.mathworks.com/matlabcentral/answers/1118-parallel-simulation, on July 27, 2011.

McGregor, D.W. & Cain, M.J. (2011). An Introduction to SIMUL8. Retrieved from http://www.mang.canterbury.ac.nz/courseinfo/msci/msci680/mcgregor_cain2004/webpage/SIMUL8.pdf, on July 27, 2011.

Michalowski, P. (2009), *Tyche 2.3.* DRDC CORA Report, CR 2009-005.

Microsoft, (2008a), *Windows HPC Server 2008 – Technical Overview of Windows HPC Server 2008.* Retrieved from http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=5376, on June 10, 2011

Microsoft, (2008b), *Windows HPC Server 2008 – Using Windows HPC Server 2008 Job Scheduler.* Retrieved from http://www.clustervision.com/Windows_HPC_Server_2008_Job_Scheduler.pdf, on June 1, 2011.

Microsoft, (2010a), *Windows HPC Server 2008 R2 – Technical Overview of Windows HPC Server 2008 R2 – White Paper.* Retrieved from http://www.microsoft.com/downloads/en/details.aspx?FamilyID=78adb249-e3a7-4961-b5eb-042216209023, on June 1, 2011.

Microsoft, (2010b), *Windows HPC Server 2008 R2 – Using Windows HPC Server 2008 R2 Job Scheduler*. Microsoft Corporation. Retrieved from http://social.technet.microsoft.com/Search/en-US?query=R2%20Job%20Scheduler&beta=0&ac=8, on June 1, 2011.

Microsoft, (2011), *Windows HPC Server 2008 R2 Suite.* Retrieved from http://www.microsoft.com/hpc/en/us/default.aspx on June 9, 2011.

Morin. (2011). Managed C# versus unmanaged C++. Retrieved from http://www.kbcafe.com/articles/CSharp.Performance.pdf, on Oct. 3, 2011.

Mottelet, S. & Pauss. A. (2007). XMLlab: multimedia publication of simulations applets using XML and Scilab. Retrieved from http://arxiv.org/PS_cache/arxiv/pdf/1102/1102.5711v1.pdf, on August 4, 2011.

Nicol, D. & Fujimoto, R. (1994). Parallel simulation today. *Annals of Operations Research,* 53:249-286.

Nicol, D.M., Greenberg, A.G. & Lubachevsky, B.D. (1994). Massively parallel algorithms for trace-drive cache simulations. *IEEE Transactions on Parallel and Distributed Systems,* 5(8):849-858.

Nicol, D. & Liu, J. (2002). Composite synchronization in parallel discrete-event simulation. *IEEE Transactions on Parallel and Distributed Systems,* 13(5):433-446.

Ören, T. (2002). Growing Importance of Modelling and Simulation: Professional and Ethical Implications. *Proceedings of the 5th Conference on System Simulation and Scientific Computing (ICSC'2002).* Retrieved from http://www.site.uottawa.ca/~oren/pubs/2002/07-Shanghai.pdf, on July 19, 2011.

Page, Jr, E.H. (1994). *Simulation modeling methodology: principles and etiology of decision support.* Thesis of Ph.D. of Virginia Polytechnic Institute and State University. Retrieved from http://www.thesimguy.com/articles/simModMeth.pdf, on June 10, 2011.

Paquet, J. (2011). Programming languages performance ranking. Retrieved from http://newton.cs.concordia.ca/~paquet/wiki/index.php/Programming_languages_performance_ranking, on Sep. 28, 2011.

Pegden, C.D. & Sturrock, D.T. (2010).Introduction to Simio. *Proceedings of the 2010 Winter Simulation Conference.* Retrieved from http://www.informs-sim.org/wsc10papers/001.pdf, on July 27, 2011.

Pegden, C.D. (2011a). An Introduction to Simio for Beginners. Retrieved from http://www.simio.com/resources/white-papers/Introduction-to-Simio/introduction-to-simio-for-beginners.htm, on July 27, 2011.

Pegden, C.D. (2011b). Intelligent Objects: The Future of Simulation. Retrieved from http://www.simio.com/resources/white-papers/Intelligen-objects/Intelligent-Objects-The-Future-of-Simulation-Page-1.htm, on July 27, 2011.

Perumalla, K.S. (2006). Parallel and distributed simulation: tradition techniques and recent advances. *Proceedings of the 2006 Winter Simulation Conference (WSC'06),* 84-95, December, Monterey, CA, USA.

Prassad, S. & Deo , N. (1991). An efficient and scalable parallel algorithm for discrete-event simulation. *Proceedings of the 1991 Winter Simulation Conference (WSC'91),* 652-658, Phoenix, AZ, USA.

Ptolemy. (2011). Ptolemy II. Retrieved from http://ptolemy.berkeley.edu/ptolemyII/, on August 3, 2011.

Rizzoli, A.E. (2011). A collection of Modelling and Simulation Resources on the Internet. Retrieved on July 12, 2011, from http://www.idsia.ch/~andrea/sim/simtools.html.

Rockwell Automation. (2011). Arena Simulation Software. Retrieved from http://www.arenasimulation.com/, on July 27, 2011.

Saker Solutions. (2011). WITNESS. Retrieved from http://www.sakersolutions.com/sakerftp/Witness%20downloads/Witness.pdf, on August 3, 2011.

Schriber, T. J. & Brunner, D. T., (2010), Inside Discrete-Event Simulation Software: How It Works and Why It Matters. *Proceedings of the 2010 Winter-Simulation Conference*, December, 2010, Baltimore, MD, USA.

Sen, R. (2008), *Developing Parallel Programs. Microsoft Corporation.* Retrieved from http://msdn.microsoft.com/en-us/library/cc983823.aspx, on June 1, 2011.

Sestoft, P. (2010). Numeric performance in C, C# and Java. Retrieved from http://www.itu.dk/~sestoft/papers/numericperformance.pdf, on Sep. 30, 2011.

Simio. (2011a). Enterprise Data In, Enhanced Output Analysis Out. Retrieved from http://www.simio.com/about-simio/why-simio/enterprise-data-in-enhanced-output-analysis-out.html, on July 27, 2011.

Simio. (2011b). The Future of Simulation, Growing with You. Retrieved from http://www.simio.com/about-simio/why-simio/simio-is-the-future-of-simulation-software-growing-with-you.html on July 27, 2011.

Simio. (2011c). Simio Simulation Blog. Retrieved from http://blog.kimesolutions.de/2010/05/beware-of-simios-parallelism_16.html and http://blog.kimesolutions.de/2010/05/beware-of-simios-parallelism_16.html, on July 27, 2011.

Simul8. (2011a). Parallel Processing. Retrieved from http://www.simul8.com/support/help/doku.php?id=features:parallel_processing, on July 27, 2011.

Simul8. (2011b). Using SIMUL8 across Networks. Retrieved from http://www.simul8.com/support/newsletter/UsingSIMUL8acrossNetworks.htm, on July 27, 2011.

Sulistio, A., Yeo, C.S., & Buyya, R. (2002). Simulation of Parallel and Distributed Systems: A Taxonomy and Survey of Tools. *International Journal of Software Practice and Experience*, 1-19, Wiley Press. Retrieved from http://ww2.cs.mu.oz.au/~raj/papers/simtools.pdf, on June 13, 2011.

Sulistio, A., Yeo, C.S. & Buyya, R. (2004). A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. Software – Practice & Experience, 34(7):653-673.

Taylor, S.J.E., Winter, S.C. & Wilson, D.R. (1993). A methodology for the decomposition of discrete event models for parallel simulation. *Proceedings of the Parallel and Distributed Processing*, 536-543, January, Gran Canaria, Spain. Retrieved from http://bura.brunel.ac.uk/bitstream/2438/3354/1/A%20methodology%20for%20the%20decomposition.pdf, on June 10, 2011.

Thuyet, D.Q. (2010). Simulation Tools Software. Retrieved from http://dangquothuyet.110mb.com/index.php?news&nid=4, on July 26, 2011.

Tropper, C. (2002). *Parallel and Distributed Discrete Event Simulation.* Nova Science Pub Inc. Also retrieved from http://books.google.ca/books?id=TC2cLuLu-XIC&pg=PR7&lpg=PR7&dq=parallel+and+distributed+discrete+event+simulation,+carl+tropper&source=bl&ots=W3qiwu2F59&sig=iHY9q_3Ota0-Z8lhbEJVcglNMQ4&hl=en&ei=kzr2TYmNKaXw0gG5-fXtDA&sa=X&oi=book_result&ct=result&resnum=2&ved=0CCsQ6AEwAQ#v=onepage&q&f=false, on June 13, 2011.

Tommti-systems. (2011). Performance comparison C++, C# and Java. Retrieved from http://www.tommti-systems.de/go.html?http://www.tommti-systems.de/main-Dateien/reviews/languages/benchmarks.html, on Sep. 30, 2011.

Trumphurst. (2011). Available C++ libraries FAQ. Retrieved on July 11, 2011, from http://www.trumphurst.com/cpplibs/datapage.php. Trumphurst.com.

Twistedmatrix. (2011). A subjective analysis of two high-level, object-oriented languages. Retrieved from http://twistedmatrix.com/~glyph/rant/python-vs-java.html, on Sep. 30, 2011.

Tyche Software Helper. (2011). Tyche 2.3 Software – Help – Tyche Programmer's Reference – Shared Code Modules – Simulation Engine.

VBForums. (2002). C# and VB.NET benchmark comparison. Retrieved from http://www.vbforums.com/showthread.php?t=169894, on Oct. 3, 2011.

Vee, V.-Y. & Hsu, W.-J., (1990), Parallel Discrete Event Simulation: A Survey. *Information Systems Journal*, 1-30.

Volkerschatz. (2011). A comparison of programming languages and techniques for a combinatorial problem. Retrieved from http://www.volkerschatz.com/science/nonpapers/mannwhitney.html, on Oct. 3, 2011.

Wainer, G.A. (2011). DEVS Tools. Retrieved on July 11, 2011, from http://www.sce.carleton.ca/faculty/wainer/standard/tools.htm.

Wang, X., Lei, Z., Nong, Z. & Tang, Y. (2009). Time management in parallel discrete event simulation. *Proc. of the International Forum on Information Technology and Applications (IFITA '09)*, pp. 209-212, Chengdu, China.

Wilkinson, D. (2011). Gibbs sampler in various languages (revisited). Retrieved from http://darrenjw.wordpress.com/2011/07/16/gibbs-sampler-in-various-languages-revisited/, on Sep. 29, 2011.

Wikipedia. (2011a). List of computer simulation software. Wikipedia. Retrieved on July 11, 2011, from http://en.wilipedia.org/wiki/List_of_computer_simulation_software.

Wikipedia. (2011b). Simulation language. Retrieved on July 11, 2011, from http://en.wikipedia.org/wiki/Simulation_language.

Wikipedia. (2011c). List of discrete event simulation software. Retrieved on July 12, 2011, from http://en.wikipedia.org/wiki/List_of_discrete_event_simulation_software.

Wikipedia. (2011d). DEVS. Retrieved on July 12, 2011, from http://en.wikipedia.org/wiki/DEVS.

Wikipedia. (2011e). Category: Simulation programming languages. Retrieved on July 12, 2011, from http://en.wikipedia.org/wiki/Category:Simulation_programming_languages.

Xan. (2007). Microbenchmark on Java and C Matrix Multiplication. Retrieved from http://www.forthgo.com/blog/2007/06/21/microbenchmark-on-java-and-c-matrix-multiplication/, on Sep. 29, 2011.

Xj Technologies. (2010). AnyLogic 6.5.1 New Features. Retrieved from http://www.xjtek.com/files/AnyLogic-6.5.1-NewFeatures.pdf, on July 27, 2011.

Xj Technologies. (2011). Overview of AnyLogic. Retrieved from http://www.xjtek.com/anylogic/overview/, on July 27, 2011.

Young, M.J. (2011). Three Innovative Parallel Discrete Event Simulation Languages. Retrieved on July 12, 2011, from http://www.mjyonline.com/SimulationLanguages.htm.

This page intentionally left blank.

Appendix A: List of Libraries/Software/Tools/Lanauges for DES and PDES

(Note: There are 351 tools; the numbers in parentheses are the times of citation in survey, review and summary documents)

20-sim (3)

ACSL Sim / acslX (3)

ACTNET (1)

Add-on library BG V2.1 (1)

ADEVS (3)

AERO (1)

AMESim (4)

AnyLogic (6)

APES (1)

APOSTLE (1)

Archsim (1)

Arena (8)

ASCEND (3)

ATM-TN (1)

Aurora (1)

AUTO (1)

AutoMod (5)

Awesim (2)

BaseSim (2)

Berkeley Madonna (2)

Bounce (1)

Brahms (1)

Bricks (1)

Broadcast (1)

BuildSim (2)

C++SIM (5)

CAFTA (1)

CAMP-G (2)

Care Pathway (1)

CARMS (1)

CD++ (3)

CellSim (1)

CESIMO (1)

Chant (1)

Chemical Workbench (1)

CircuitLogix (1)

ClusterSim (1)

Cncl (1)

Cnet (2)

COCO Simulator (1)

Comnet III (1)

COMSOL (1)

CosiMate (2)

CoSMo-Sim (1)

CPSim (1)

Create (1)

CSIM17 / CSIM 19 (4)

Delmia (2)

Delsi (1)

DESIRE (3)

DESMO-J (3)

DEUS (1)

DEVS++ (1)

DEVS# (1)

DEVS/C++ (3)

DEVS/HLA/CORBA (2)

DEVS/Java (3)

DEVSim++ (2)

DEVSimPy (1)

DEx (3)

DOSIMIS-3 (1)

DqDBsim 802. (2)

DSHPlus (2)

DSM (2)

DSOL (3)

DX Studio (1)

Dymola (4)

Dynawiz (2)

EASY5 (2)

Ecolego (3)

EcosimPro (3)

EICASLAB (1)

ED simulator (1)

EJS (1)

eM-Plant X (2)

Enterprise Dynamics (3)

Extend (1)

ExtendSim (4)

Faber (1)

Facsimile (3)

Flexsim (6)

Flowmaster (1)

FoCs (2)

Forio (1)

FreeMat (3)

G2 Platform (1)

Galatea (5)

GarlicSim (2)

GASP (1)

Gepasi (1)

GK-DEVS (1)

Gnans (1)

GloMoSim (3)

GoldSim (6)

gPRPOMS (2)

GPSS/H (8)

Grid Matrix (1)

GridSim (4)

GSPN (1)

GTNetS (2)

GTW (Georgia Tech Time Warp) (2)

GUS (1)

HCADwin (2)

HighMAST (2)

HIRES (1)

ICMS (2)

iGrafx Process (1)

IMTEK (2)

Isaac (1)

Ithink (2)

ITI-SIM (2)

JAMES II M & S Framework (1)

Java Modelling Tools (1)

JavaSIM (3)

JDEVS (4)

JiST (4)

JMadonna (1)

JModelica (1)

JSDESLib (1)

JSIM (3)

JSL (1)

Khimera (1)

LabVIEW (2)

LBNL NS Simulator (1)

Lean Modeler (2)

L-SIM (1)

LSIS DME (2)

Maisie/MOOSE (2)

Maple (1)

MapleSim (1)

MaRS (1)

MASON (1)

MAST / OpenMAST (1)

MathCore (3)

Mathematica (2)

Mathtools (2)

MATLAB (2)

MATRIXx (1)

MATSIM (1)

MCSim (1)

MicroGrid (1)

Micro Saint /sharp (2)

Mimosa (1)

Mirelle (1)

Mist (1)

MJX (1)

ML Designer (2)

MMS (2)

MobSim (1)

Modelica (3)

ModelMaker (2)

MODSIM III (4)

Monkeysim (1)

Mosis (1)

MS1 (2)

MS-MPI with Visual Studio (2)

MTT (2)

Must (1)

myM visual simulation tool (1)

mystrategy (2)

NAG (2)

NCTuns (2)

NEi Nastran (1)

NEST (2)

Netlib (2)

NetLogo (2)

NetSim (5)

NETWORK II.5 (2)

NI Multisim (1)

NI MATRIXx (2)

NIST ATM (1)

NS2/NS3 (4)

Octave (3)

OMNeT++ (2)

OpenCL Studio (1)

OpEMCSS (2)

OpenModelica (1)

OPNET-Modeler (2)

OptSim (2)

ORCA (1)

Os4Mod (1)

PADE (1)

PARALLAXIS (1)

ParaSol (4)

PARL (1)

Parsec (5)

ParView (1)

Pasion (1)

PDNS (1)

PDP++ (1)

PEPSY-QNS (1)

Performance PROPHET (2)

Petri Nets Tools list (1)

PF3S (2)

Physics Abstraction Layer (1)

PhySim (2)

PLaneT / PLT (1)

PlantSimulation (5)

POPULUS (1)

Portable Run Time System (1)

Portfolio simulator (1)

POSE (1)

Poses++ (1)

PowerDEVS (3)

PowerSim (3)

PowerXplorer (1)

Preactor 2000 (1)

Processmodel (1)

Process simulator (1)

ProDyn (1)

profiSEE (1)

PRO/II (1)

Project simulator (1)

ProModel (4)

ProperCAD (1)

Prophesy (2)

Ps-I (2)

PSK (1)

PSM++ (1)

Ptolemy (5)

Python DEVS (2)

QNAP2 (1)

QSIM (3)

QSNAP (2)

QualNet (2)

QUEST (1)

Quick Threads (1)

QX3D (2)

REAL Network Simulator (2)

Reno (1)

Renque (3)

ReThink (1)

RoboLogix (1)

ROSS (1)

RT-LAB (1)

SansGUI (2)

SCALE RT (1)

Schedula (1)

Scicos (4)

Scilab (4)

SDX (2)

SEMoLa (2)

SeSAm (4)

SES-Workbench (3)

SharpSim (1)

Shift (3)

ShowFlow (2)

Shunra VE (2)

SIGMA (2)

Silk (1)

Sim++ (1)

SIMAN/CinemaV (1)

SimApp (1)

SIMAS II (2)

SIMBAX (2)

SimBeans (2)

SimC (1)

SimCAD Pro (4)

SIMCOS (1)

SimCreator (2)

SimEvents (2)

SIMEX (2)

SimFlex (2)

SimGrid (3)

Simgua (1)

SIMIC (1)

Simile (3)

Simjava (5)

Simio (5)

Simkit (6)

SimLab (3)

SIMLOX (1)

SimOS (3)

SimPack (3)

Simple 1 (2)

SimPLE++ (2)

SIMPLORER (2)

SimPlusPlus (1)

SIMPROCESS (3)

SimPy (8)

SIMSCRIPT II.5, III (7)

SimTools (1)

SIMUL8 (6)

Simula (5)

Simulacon4 (2)

SimulAr (2)

Simulation 123 (2)

Simulation 8000 (1)

Simulation Plus (1)

Simulation X (2)

Simulink (4)

SimWalk (2)

SimWiz (1)

SINA (1)

SLAM (1)

SLX (3)

SmallDEVS (4)

SMPL (1)

SPaDES (3)

SPAR (2)

SPEEDES (1)

Spice (2)

SRGSim (1)

SSS (2)

Stampack (1)

STARDIS (2)

StarLogo (1)

StateFlow (1)

STELLA II (4)

SUMO (1)

SWANS (2)

Swarm-SantaFe Institute (2)

Symbols (2)

Tarsier (2)

Taylor II (2)

TIERRA (1)

TomasWeb (2)

TOOPS (2)

Tortuga (3)

TT / Model Transformation Tools (1)

Traffic (2)

TRUE (3)

TWOS (3)

uC++ (1)

Universal Mechanism (2)

UPS (1)

µSik (1)

uSystem, muC++ (1)

Vensim (5)

Verilog-AMS (1)

VHDL (2)

VisSim (5)

Visual Components (2)

Visualsim (1)

VLE (2)

VSTEP (1)

Vulcan – FEM (1)

WARPED (7)

WinA&D (2)

WinSAAM (2)

Appendix B: Top 10 libraries/software/tools/lanauges for DES and PDES

Name	Description	Parallelization	API/Lib/ language	Language of implementation	OS	GUI / Visual	License	Developer	Applications	Notes	References
AnyLogic 6.6 (Jul-11)	Support three modelling approaches: (1) System dynamics, (2) Discrete event simulation and (3) Agent-based modeling.	Support multiple runs in parallel on a multi-core computer	Integrated tool with GUI; Can extend with Java code; Drag from palette; 2D/3D views	Java SE	Cross - platf orm	GUI, Visual, animati on	Proprie-tary	XJ Technolog ies (initially developed at Saint Petersburg Technical Univ.)	General: Finance, business, dynamics, healthcare, transportation, logistics, manufacturing, social dynamics, supply chains, urban	(1) Supporting multi-simulation runs on multi-cores in parallel; (2) Java-based; able to extend with Java code, but no API found; (3) Good support (4) Commercial; (5) 100 + customers (6) The multi-method is good for multiple application domains, but complicated functionality, in particular AI functions, may affect efficiency.	http://en.wikipedia.org/wiki/AnyLogic; www.xjtek.com; (xj Technologies, 2011; xj Technologies, 2010)
Arena 13.9 (2010).	A discrete event simulation software	No "Parallelizatio n" support found; Running on multi-cores/process ors, but only one instance of Arena at a time	Integrated tool with GUI No API found	Visual Basic; Integrating very well to VB	Wind ows 7/Vis ta/XP	GUI, OpenG L	Comme rcial Edition, & Acade mic Edition	Originally Cambridg e UK, Systems Modeling and acquired by Rockwell Automatio n in2000.	General. Business process by GM, UPS, IBM, Nike, Zerox, Ford.	http://en.wikipedia.org/wiki/Arena_(software); http://www.arenasimulation.com/; (Rockwell, 2011; Horejsi, 2011)	
GridSim V5.2 (Nov-10)	A toolkit used for modeling and simulation of entities in parallel and distributed computing	Support parallelization	API	Java	Cross - platf orm	?	GPL	Univ. of Melbourn e	General: Biotechnology, astrophysics, network design, high-energy physics, scientific,	(1) Grid computing focuses on more loosely coupled, heterogeneous and geographically dispersed. Cluster	http://www.cloudbus.org/gridsim/; (Buyya & Marshed, 2002)

Name	Description	Parallelization	API/Lib/ language	Language of implementation	OS	GUI / Visual	License	Developer	Applications	Notes	References
	(PDC) systems								engineering.	computing uses similar /identical computers & OS (2) Grid computing is suitable for non-communication parallel applications (3) No publically identifiable customer on the software web site	
Microsoft MPI /OpenMP	A standard programming library	Parallel: MPI, OPMP	API for C# & VC++	VC++	Windows	n/a	Microsoft	Microsoft	General	(1) Visual studio compatible (2) Can run on Windows HPC Server 2008	Sen, 2008
PARSEC 1.1 (2011?)	Parallel Simulation Environment for Complex Systems.	Parallel (MPI, PVM, P4)	PARSEC language functions	PARSEC Language: C-based	Solaris, Linux Redhat 6.x, Win NT/2k, AIX, Irix6.	n/a	Own license	Parallel Computing Laboratory at UCLA	General: VLSI, networks, file management	(1) Good documents (2) A specific C-like programming language (3) Support parallelization (4) No publically identifiable customer on the software web site	Cambridge SRG, 2011; http://may.cs.ucla.edu/projects/parsec/
Ptolemy II 8.0.1 (Oct-10)	A framework supporting experimentation with actor-oriented design	Multi-thread based; can execute multiple actors in parallel (e.g. on a multi-core machine)	Framework	Java	Cross-platform	GUI & visualization	Fairly liberal UC Berkeley copyright	UC Berkeley	General: network discrete-events, dataflow	(1) Actors (software components) communication via interconnected ports, rather than supporting multi-	Sulistio et al., 2004; http://ptolemy.eecs.berkeley.edu/ptolemyII/; (Ptolemy, 2011; Lee & Neuendorffer, 2007)

Name	Description	Parallelization	API/Lib/ language	Language of implementation	OS	GUI / Visual	License	Developer	Applications	Notes	References
										cluster nodes with multi-cores. (2) Using multi-threads to support parallelization (3) Not updated for a while (4) Over 10 customers found (5) Research focused	
SimEvents R2010b (Sep-10) in SimuLink - Matlab	Used to model process flows and logistics in order to understand resource availability, inventory management techniques, and the effects of arbitrary events on a mission plan	Can support multiple runs of a SimEvents model with MatLab Parallel Computing Toolbox	Integrated Tool with GUI; Library blocks for writing algorithms to customize operations such as routing, processing delays, and prioritization	MATLAB in C, Java	Cross - platform	GUI; Library of blocks for modeling the motion of entities	Proprietary	MathWorks	General: electronic system, process/logistics, hybrid simulations	(1) Extension to SimuLink (2) MatLab "Parallel Computing Toolbox" can support multiple runs of a SimEvents model in parallel. (3). Commercial (4). Capable of customizing (5). Some kind of stronger support to industrial control process, e.g. routing, switching and gating entities. (6) Good interface to time-based dynamics of SimuLink. (1). 1 customer found	http://en.wikipedia.org/wiki/SimEvents; (Mathworks, 2011a, 2011b)
Simio three (Apr-11)	A unique multi-paradigm simulation software tool that provides a rapid and flexible	(1) Auto multi-core support ; (2) users can expand its API with .NET C#	Stand alone tool; Allowing subclass a standard library object for	V C# or VB .Net	Windows	GUI; Can display 3-D animation	Commercial	Simio LLC	General: advanced analytics, airports, manufacturing, supply chain, healthcare,	(1) Auto multi-core support (2) C# based; User can expand with .NET languages e.g. C# or VB	http://www.simio.com/index.html; (Pegden, 2010, 2011a, 2011b; Simio, 2011a, 2011b, 2011c)

Name	Description	Parallelization	API/Lib/ language	Language of implementation	OS	GUI / Visual	License	Developer	Applications	Notes	References
	modeling capability without requiring programming	or VB	customization; API						military, mining, ports, predict process variability	(3) 4 customers found (4) Used in over 300 universities for teaching	
Simjava 1.2	A process based discrete event simulation package for Java, similar to Jade's Sim++, with animation facilities	Multi-threads: each simulation is a thread	lib	Java, C++	Cross-platform	Vrml components, with animation facilities	?	Univ. of Edinburgh	General: process-based DES package with animation facilities; did not find application area description	(1) Support multi-threads; not support MPI (2) 11 customers found (3) No good documents found	Constitution, 2011; http://www.dcs.ed.ac.uk/home/hase/simjava/
SimPy 2.1.0 (Jun-10)	A DES package based on Python. A process-based, OO discrete-event simulation language	Parallel with parallel Python	API	Python	Cross-platform; Need Parallel Python installed	GUI for debugger	GPL	SimPy developer community: senior Students of Norm Matloff at UofCali at Davis	General: epidemics, traffic, air space, industrial engineering, computer hardware, workflow	(1) Need parallel Python installed for parallelization (2) 25 users found (3) Need to write Python code	Wikipedia, 2011a; http://simpy.sourceforge.net/
SIMUL8 V2010	Software for simulating systems that involves processing of discrete entities at discrete times	Can spread the runs of simulation model between two or more networked computers where you have installed SIMUL8	Integrated Tool, Drag-and-drop or using Visual Logic VB-like language	Visual Basic based. Users may use Visual Logic VB-like language	Windows NT, 95, 98, 2000, XP & Vista	GUI	Proprietary	SIMUL8 Corporation	General: manufacturing, business, healthcare, public sector, supply chain, logistics, energy, justice, call centers	(1) Parallelizing multiple runs with networked computers, but using only spare CPU time: no balanced loading, not priorities (2) All computers for parallel execution must install SIMUL8 (3) Need to learn Visual Logic VB-like language	http://en.wikipedia.org/wiki/Simul8; http://www.simul8.com/; (McGregor & Cain, 2011; Simul8, 2011a, 2011b)

Name	Description	Parallelization	API/Lib/ language	Language of implementation	OS	GUI / Visual	License	Developer	Applications	Notes	References
										(4) Suitable for military applications (5) Good documents and support (6) 35 customers found	

Appendix C: Results of functionality reveiw of simulation development environments

Simulation Development Software Functionality Checklist: Visual Basic

The following summarizes the features that are necessary in any programming package to be used in the redevelopment and further new development of the Tyche simulation engine. The features are presented in the following sections in a tabular format with "Yes/No" columns to ease in identifying the software package's fit to requirements. Additional comments may be added as necessary (for example, if the answer were no, but the package could accommodate the functionality in a different manner).

VB: The Operating Environment

Over and above the programming language, there are a number of aspects of the operating environment that are significant to the effective development of a software project. The following should be considered (no particular order).

Requirement	Yes	No	Additional Comments
Native Code Compilation - Even in a multi-processing environment, processing speed is an important concern if we wish to "play nice" with other users of the hardware. Interpreted languages and languages that compile to p-code will not execute quickly enough.	Yes		Using P-Code prior to version 5. http://en.wikipedia.org/wiki/Visual_Basic
Error Handling - The existing simulation software contains extensive error handling code. Along with better pinpointing of problems, error handling also ensures that the simulation is able to recover from errors and continue processing with minimal loss of data and time. Runtime errors will happen. If the new system were to simply abort on an error (losing unsaved data) or only provided an end-state information dump, this would be unacceptable.	Yes		VB supports structured exception (error) handling, which allows the program to detect and possibly recover from error during execution. http://msdn.microsoft.com/en-us/library/s6da8809(v=vs.80).aspx Try…Catch…Final
Symbolic Debugging (Debugging Environment) - Further to the "Error Handling" comments above, the complexity of the code dictates a need for a modern software-debugging environment (comparable to what has been provided in most mainstream development tools since the mid-1990s). Errors can be difficult to isolate and a proper debugging environment will assist in addressing problems promptly.	Yes		http://msdn.microsoft.com/en-us/library/ms172744(v=VS.80).aspx
Native Disk I/O Support - Though disk I/O is not currently performed during the iterative process in the current simulation, it is performed in the setting up of the simulation and in the reporting of results. Native disk I/O functions (as opposed to I/O through a third-party mechanism) performs significantly better. It is also possible that the approach to I/O may change as a consequence of multiprocessing. Results generated in each iteration might need to be written immediately. This would make native disk I/O even more important. Native disc I/O refers to reading and writing at the file, folder and drive levels.			
Native File I/O	Yes		http://support.microsoft.com/kb/304427
Native Folder I/O	Yes		http://msdn.microsoft.com/en-us/library/9wzc03ze(v=vs.80).aspx

Requirement	Yes	No	Additional Comments
Native Drive I/O	Yes		http://msdn.microsoft.com/en-us/library/9wzc03ze(v=vs.80).aspx
Command Line Support – The current simulation can be executed directly from a command line. The run's XML file name is passed through a command parameter. This functionality should still be available to the user.	Yes		
Spawned Processing – Toward the end of the simulation run, the output file is compressed through the use of the WinZip Command Line Interface. This is accomplished through the dynamic spawning of a separate command process.	Yes		http://www.developerfusion.com/code/2043/winzip-command-line-information/

VB: Data Handling

The simulation uses a rich and varied set of data structures to represent its configuration and the objects that are manipulated. During the running of the model, its state is determined by the data content of its objects. The new software development platform must provide data structuring mechanisms sufficient to define and store the data objects needed. The following sections outline those needs

Requirement	Yes	No	Additional Comments
Record Definitions – In addition to the standard scalar variable types (string, integer, long, Boolean etc.), programmers must have a means of defining data structures that aggregate several different variables together so that data objects with multiple attributes can be created. At a minimum, a record definition mechanism must be provided. The current model uses more sophisticated data objects with properties and methods associated with them. Ideally, something similar to these "class" objects should also be supported.			
Scalar types that must be supported include:			
Byte	yes		
Boolean	yes		
Integer	yes		
Long	yes		
Single	yes		4 bytes float
Double	yes		8 bytes float
Date	yes		8 bytes
String	yes		class
More complex structures include:			
Record Definitions	yes		e.g. UserDefined, Object/class
Complex Data Objects (Classes)	yes		e.g. object/class

Requirement	Yes	No	Additional Comments
Dynamic Collections – Data collections are the heart and soul of the current simulation system. A collection mechanism provides for the dynamic grouping of data objects of common structure. Collections grow and shrink as the program proceeds. The collection mechanism must support adding and removing members dynamically. The number of items in a collection must be able to be determined at any time and the items in a collection must be able to be accessed and updated. Collection support is essential to any development tool to be selected. Require properties include:			
Adding/Removing items dynamically	yes		e.g. Collection/list
"Count" property	yes		e.g. Collection/list
Dynamic update of items	yes		e.g. Change attributes of an item in a collection
Size limited only by available memory		no	The index of an Array cannot be greater than the maximum of a 32-bit value(2147483648). In reality though, the app would run out of memory first. http://vbnet.mvps.org/index.html?http://vbnet.mvps.org/dev/main/vb6faq.htm
Variable Scoping – The idea that a variable may have limitations to its scope has been pretty well accepted for quite some time. However, some languages (most notably COBOL) don't support variable scoping. Since nothing should be taken for granted with respect to what is supported in special purpose software development tools, variable scoping (localization) should be supported. A single global data pool would be too cumbersome. Variables should be able to be scoped as:			
Global	yes		http://support.microsoft.com/kb/141693
Modular	yes		http://support.microsoft.com/kb/141693
Procedural	yes		http://support.microsoft.com/kb/141693

VB: Processing Constructs

Requirement	Yes	No	Additional Comments
Modules – Modules allow code in large applications to be broken up logically into smaller and more manageable units. Some means of logically arranging program code must be supported in any development tool to be selected.	yes		
Procedures and Functions – Procedures and functions provide a means of dividing up processing logic into portions that can be called repeatedly. The programming language used by the new development tool must be procedural.	yes		
The following procedure types should be supported:	yes		
Subroutines	yes		
Functions			

Requirement	Yes	No	Additional Comments
Both of these constructs should support parameter passing as follows:	yes		Ref: http://msdn.microsoft.com/en-us/library/ddck1z30.aspx
By reference By Value	yes		Ref: http://msdn.microsoft.com/en-us/library/ddck1z30.aspx
Parameters should be either of the scalar types noted in the previous section and	yes		
User defined record structures and Complex data objects	yes		object
Looping Structures – Most of the processing conducted in the model iterations involve looping structures of several types working within complex conditional processing. Both pre-test (FOR … NEXT, DO WHILE) and post-test (REPEAT … UNTIL) loop constructs are needed. Loops must also be able to be nested within each other. Looping structures should include the following:			
FOR … NEXT	yes		For…next
DO WHILE	yes		Do while…loop; do …loop while/until
REPEAT … UNTIL	yes		Do …loop while/until
Conditional Constructs – As mentioned above, there is within the model a great amount of complex conditional processing. The programming language must support both IF… THEN … ELSE and CASE constructs. These structures must also be able to be nested within each other. The following conditional constructs must be supported:			
IF … THEN … ELSE	yes		If…elseif…else…end if
CASE statements	yes		Select case…case…end select
In specifying conditions, complex logical constructs must be supported	yes		A condition requires a logic expression.
Boolean functions should be able to be used within logical conditions	yes		And, Or, AndAlso, OrElse, Xor, Not, <, >, <=, >=

VB: Miscellaneous Support

Requirement	Yes	No	Additional Comments
MS Excel Support – The current simulation provides some of its output in a Microsoft Excel workbook. The model directly supports outputting of information to Excel. This level of support isn't strictly needed as long as the necessary output can be formatted as either comma-delimited or tab-delimited text.	yes		e.g. with Visual Studio .Net
Random Number Generation – The current simulation uses random numbers extensively. The new development tool will need to be able to provide random numbers.	yes		New Random (lowerbound, upperbound)
The seed value for the random number generator will have to be adjustable programmatically.	yes		New Random (lowerbound, upperbound)

Simulation Development Software Functionality Checklist: Visual C++

The following summarizes the features that are necessary in any programming package to be used in the redevelopment and further new development of the Tyche simulation engine. The features are presented in the following sections in a tabular format with "Yes/No" columns to ease in identifying the software package's fit to requirements. Additional comments may be added as necessary (for example, if the answer were no, but the package could accommodate the functionality in a different manner).

VC++: The Operating Environment

Over and above the programming language, there are a number of aspects of the operating environment that are significant to the effective development of a software project. The following should be considered (no particular order).

Requirement	Yes	No	Additional Comments
Native Code Compilation - Even in a multi-processing environment, processing speed is an important concern if we wish to "play nice" with other users of the hardware. Interpreted languages and languages that compile to p-code will not execute quickly enough.	yes		
Error Handling - The existing simulation software contains extensive error handling code. Along with better pinpointing of problems, error handling also ensures that the simulation is able to recover from errors and continue processing with minimal loss of data and time. Runtime errors will happen. If the new system were to simply abort on an error (losing unsaved data) or only provided an end-state information dump, this would be unacceptable.	yes		Try…catch…throw
Symbolic Debugging (Debugging Environment) - Further to the "Error Handling" comments above, the complexity of the code dictates a need for a modern software-debugging environment (comparable to what has been provided in most mainstream development tools since the mid-1990s). Errors can be difficult to isolate and a proper debugging environment will assist in addressing problems promptly.	yes		http://msdn.microsoft.com/en-us/library/k70yt3e2%28v=vs.71%29.aspx
Native Disk I/O Support - Though disk I/O is not currently performed during the iterative process in the current simulation, it is performed in the setting up of the simulation and in the reporting of results. Native disk I/O functions (as opposed to I/O through a third-party mechanism) performs significantly better. It is also possible that the approach to I/O may change as a consequence of multiprocessing. Results generated in each iteration might need to be written immediately. This would make native disk I/O even more important. Native disc I/O refers to reading and writing at the file, folder and drive levels.			
Native File I/O	yes		http://support.microsoft.com/kb/307398
Native Folder I/O	yes		http://msdn.microsoft.com/en-us/library/system.io.directory.createdirectory%28v=vs.71%29.aspx
Native Drive I/O	yes		http://social.msdn.microsoft.com/Forums/en-US/vcgeneral/thread/7c8f6ab7-8625-4829-8cdb-3c304752c62c/
Command Line Support – The current simulation can be executed directly from a command line. The run's XML file name is passed through a command parameter. This functionality should still be available to the user.	yes		

Requirement	Yes	No	Additional Comments
Spawned Processing – Toward the end of the simulation run, the output file is compressed through the use of the WinZip Command Line Interface. This is accomplished through the dynamic spawning of a separate command process.	yes		http://www.memecode.com/docs/winzip.html; http://stackoverflow.com/questions/3301930/how-do-i-execute-winzip-from-visual-studio-without-its-gui-opening

VC++: Data Handling

The simulation uses a rich and varied set of data structures to represent its configuration and the objects that are manipulated. During the running of the model, its state is determined by the data content of its objects. The new software development platform must provide data structuring mechanisms sufficient to define and store the data objects needed. The following sections outline those needs

Requirement	Yes	No	Additional Comments
Record Definitions – In addition to the standard scalar variable types (string, integer, long, Boolean etc.), programmers must have a means of defining data structures that aggregate several different variables together so that data objects with multiple attributes can be created. At a minimum, a record definition mechanism must be provided. The current model uses more sophisticated data objects with properties and methods associated with them. Ideally, something similar to these "class" objects should also be supported.			
Scalar types that must be supported include:			
Byte	yes		char
Boolean	yes		bool
Integer	yes		int
Long	yes		long
Single	yes		float
Double	yes		double
Date String	yes		e.g. structures: SYSTEMTIME, FILETIME, CTIme::GetCurrentTime()
More complex structures include: Record Definitions			
Complex Data Objects (Classes)	yes		struct
	yes		Object/class
Dynamic Collections – Data collections are the heart and soul of the current simulation system. A collection mechanism provides for the dynamic grouping of data objects of common structure. Collections grow and shrink as the program proceeds. The collection mechanism must support adding and removing members dynamically. The number of items in a collection must be able to be determined at any time and the items in a collection must be able to be accessed and updated. Collection support is essential to any development tool to be selected.			
Require properties include:			

Requirement	Yes	No	Additional Comments
Adding/Removing items dynamically	yes		e.g. list, insert(), remove() in linkedlist, hashset
"Count" property	yes		e.g. size()
Dynamic update of items	yes		e.g. reset attributes in an object in a list
Size limited only by available memory		no	The maximum number of elements in a collection is Int32.MaxValue – just over two billion. In practice, you'll most likely run out of allocable memory before you reach those limits, especially if you're running on a 32-bit system. Ref: http://stackoverflow.com/questions/3906891/what-is-the-max-limit-of-data-into-liststring-in-c
Variable Scoping – The idea that a variable may have limitations to its scope has been pretty well accepted for quite some time. However, some languages (most notably COBOL) don't support variable scoping. Since nothing should be taken for granted with respect to what is supported in special purpose software development tools, variable scoping (localization) should be supported. A single global data pool would be too cumbersome.			
Variables should be able to be scoped as:			
Global	yes		e.g. public classes, interfaces, or static variables
Modular	yes		Classes, source files
Procedural	yes		Functions in classes

VC++: Processing Constructs

Requirement	Yes	No	Additional Comments
Modules – Modules allow code in large applications to be broken up logically into smaller and more manageable units. Some means of logically arranging program code must be supported in any development tool to be selected.	yes		e.g. classes, source code files
Procedures and Functions – Procedures and functions provide a means of dividing up processing logic into portions that can be called repeatedly. The programming language used by the new development tool must be procedural.			
The following procedure types should be supported:			
Subroutines	yes		Implemented via functions in classes
Functions	yes		Functions in classes
Both of these constructs should support parameter passing as follows:			
By reference	yes		
By Value	yes		
Parameters should be either of the scalar types noted in the previous section and			
User defined record structures and	yes		struct
Complex data objects	yes		class

Requirement	Yes	No	Additional Comments
Looping Structures – Most of the processing conducted in the model iterations involve looping structures of several types working within complex conditional processing. Both pre-test (FOR … NEXT, DO WHILE) and post-test (REPEAT … UNTIL) loop constructs are needed. Loops must also be able to be nested within each other. Looping structures should include the following:			
FOR … NEXT	yes		For
DO WHILE	yes		while
REPEAT … UNTIL	yes		Do…while
Conditional Constructs – As mentioned above, there is within the model a great amount of complex conditional processing. The programming language must support both IF… THEN … ELSE and CASE constructs. These structures must also be able to be nested within each other. The following conditional constructs must be supported:			
IF … THEN … ELSE	yes		If…then…else
CASE statements	yes		Switch…case
In specifying conditions, complex logical constructs must be supported	yes		
Boolean functions should be able to be used within logical conditions	yes		And (&&, &), Or (|, ||), not (!), Xor(^), <<, >>, ==, <, >, <=, >=

VC++: Miscellaneous Support

Requirement	Yes	No	Additional Comments
MS Excel Support – The current simulation provides some of its output in a Microsoft Excel workbook. The model directly supports outputting of information to Excel. This level of support isn't strictly needed as long as the necessary output can be formatted as either comma-delimited or tab-delimited text.	yes		http://support.microsoft.com/kb/216686
Random Number Generation – The current simulation uses random numbers extensively. The new development tool will need to be able to provide random numbers.	yes		Random class
The seed value for the random number generator will have to be adjustable programmatically.	yes		Random(seedInt), Next(MaxInt)

Simulation Development Software Functionality Checklist: Visual C#

The following summarizes the features that are necessary in any programming package to be used in the redevelopment and further new development of the Tyche simulation engine. The features are presented in the following sections in a tabular format with "Yes/No" columns to ease in identifying the software package's fit to requirements. Additional comments may be added as necessary (for example, if the answer were no, but the package could accommodate the functionality in a different manner).

VC#: The Operating Environment

Over and above the programming language, there are a number of aspects of the operating environment that are significant to the effective development of a software project. The following should be considered (no particular order).

Requirement	Yes	No	Additional Comments
Native Code Compilation - Even in a multi-processing environment, processing speed is an important concern if we wish to "play nice" with other users of the hardware. Interpreted languages and languages that compile to p-code will not execute quickly enough.	yes		
Error Handling - The existing simulation software contains extensive error handling code. Along with better pinpointing of problems, error handling also ensures that the simulation is able to recover from errors and continue processing with minimal loss of data and time. Runtime errors will happen. If the new system were to simply abort on an error (losing unsaved data) or only provided an end-state information dump, this would be unacceptable.	yes		Try catch exception
Symbolic Debugging (Debugging Environment) - Further to the "Error Handling" comments above, the complexity of the code dictates a need for a modern software-debugging environment (comparable to what has been provided in most mainstream development tools since the mid-1990s). Errors can be difficult to isolate and a proper debugging environment will assist in addressing problems promptly.	yes		
Native Disk I/O Support - Though disk I/O is not currently performed during the iterative process in the current simulation, it is performed in the setting up of the simulation and in the reporting of results. Native disk I/O functions (as opposed to I/O through a third-party mechanism) performs significantly better. It is also possible that the approach to I/O may change as a consequence of multiprocessing. Results generated in each iteration might need to be written immediately. This would make native disk I/O even more important. Native disc I/O refers to reading and writing at the file, folder and drive levels.			
Native File I/O	yes		e.g. open(), read(), write()
Native Folder I/O	yes		e.g. createDirectory(), open(), read(), write()
Native Drive I/O	yes		e.g. GetLogicDrives()
Command Line Support – The current simulation can be executed directly from a command line. The run's XML file name is passed through a command parameter. This functionality should still be available to the user.	yes		
Spawned Processing – Toward the end of the simulation run, the output file is compressed through the use of the WinZip Command Line Interface. This is accomplished through the dynamic spawning of a separate command process.	yes		e.g. System.Diagnostics.Process(0

VC#: Data Handling

The simulation uses a rich and varied set of data structures to represent its configuration and the objects that are manipulated. During the running of the model, its state is determined by the data content of its objects. The new software development platform must provide data structuring mechanisms sufficient to define and store the data objects needed. The following sections outline those needs

Requirement	Yes	No	Additional Comments
Record Definitions – In addition to the standard scalar variable types (string, integer, long, Boolean etc.), programmers must have a means of defining data structures that aggregate several different variables together so that data objects with multiple attributes can be created. At a minimum, a record definition mechanism must be provided. The current model uses more sophisticated data objects with properties and methods associated with them. Ideally, something similar to these "class" objects should also be supported.			
Scalar types that must be supported include:			
Byte	yes		byte
Boolean	yes		bool
Integer	yes		int
Long	yes		long
Single	yes		float
Double	yes		double
Date	yes		DateTime
String	yes		string
More complex structures include:			
Record Definitions	yes		struct
Complex Data Objects (Classes)	yes		class
Dynamic Collections – Data collections are the heart and soul of the current simulation system. A collection mechanism provides for the dynamic grouping of data objects of common structure. Collections grow and shrink as the program proceeds. The collection mechanism must support adding and removing members dynamically. The number of items in a collection must be able to be determined at any time and the items in a collection must be able to be accessed and updated. Collection support is essential to any development tool to be selected.			
Require properties include:			
Adding/Removing items dynamically	yes		e.g. Collection, list, linkedList, hashtable
"Count" property	yes		e.g. Count()
Dynamic update of items	yes		e.g. changing attributes of an item in a List
Size limited only by available memory		no	The maximum number of elements in a collection is Int32.MaxValue – just over two billion. In practice, you'll most likely run out of allocable memory before you reach those limits, especially if you're running on a 32-bit system. Ref: http://stackoverflow.com/questions/3906891/what-is-the-max-limit-of-data-into-liststring-in-c

Requirement	Yes	No	Additional Comments
Variable Scoping – The idea that a variable may have limitations to its scope has been pretty well accepted for quite some time. However, some languages (most notably COBOL) don't support variable scoping. Since nothing should be taken for granted with respect to what is supported in special purpose software development tools, variable scoping (localization) should be supported. A single global data pool would be too cumbersome. Variables should be able to be scoped as:			
Global	yes		e.g. public classes, static variables
Modular	yes		e.g. classes
Procedural	yes		e.g. functions in classes

VC#: Processing Constructs

Requirement	Yes	No	Additional Comments
Modules – Modules allow code in large applications to be broken up logically into smaller and more manageable units. Some means of logically arranging program code must be supported in any development tool to be selected.	yes		classes
Procedures and Functions – Procedures and functions provide a means of dividing up processing logic into portions that can be called repeatedly. The programming language used by the new development tool must be procedural. The following procedure types should be supported:			
Subroutines	yes		e.g. calling functions of an object
Functions	yes		e.g. functions in classes
Both of these constructs should support parameter passing as follows:			
By reference	yes		Ref: http://msdn.microsoft.com/en-us/library/0f66670z%28v=vs.71%29.aspx
By Value	yes		Ref: http://msdn.microsoft.com/en-us/library/0f66670z%28v=vs.71%29.aspx
Parameters should be either of the scalar types noted in the previous section and			
User defined record structures and			
Complex data objects	yes		e.g. struct
	yes		e.g. objects
Looping Structures – Most of the processing conducted in the model iterations involve looping structures of several types working within complex conditional processing. Both pre-test (FOR … NEXT, DO WHILE) and post-test (REPEAT … UNTIL) loop constructs are needed. Loops must also be able to be nested within each other. Looping structures should include the following:			
FOR … NEXT	yes		for, foreach
DO WHILE	yes		while; do…while
REPEAT … UNTIL	yes		do…while

Requirement	Yes	No	Additional Comments
Conditional Constructs – As mentioned above, there is within the model a great amount of complex conditional processing. The programming language must support both IF… THEN … ELSE and CASE constructs. These structures must also be able to be nested within each other.			
The following conditional constructs must be supported:			
IF … THEN … ELSE	yes		if…else
CASE statements	yes		switch…case…default
In specifying conditions, complex logical constructs must be supported	yes		AND, OR, XOR, NOT, ==, <, >, <=, >=, !=, <<, shifted
Boolean functions should be able to be used within logical conditions	yes		AND, OR, XOR, NOT, ==, <, >, <=, >=, !=, <<

VC#: Miscellaneous Support

Requirement	Yes	No	Additional Comments
MS Excel Support – The current simulation provides some of its output in a Microsoft Excel workbook. The model directly supports outputting of information to Excel. This level of support isn't strictly needed as long as the necessary output can be formatted as either comma-delimited or tab-delimited text.	yes		Ref. http://support.microsoft.com/kb/302084
Random Number Generation – The current simulation uses random numbers extensively. The new development tool will need to be able to provide random numbers.	yes		e.g. Random class
The seed value for the random number generator will have to be adjustable programmatically.	yes		e.g. Random (int seed)

Simulation Development Software Functionality Checklist: AnyLogic (Java)

Summary of AnyLogic:

The functionality of AnyLogic, at its GUI level, does not meet most of the functions in the checklist. In many cases, users may use the AnyLogic GUI to develop their own models directly. However, AnyLogic is implemented in Java and users can also develop their own entities, data structures and algorithms in Java. Therefore, the following result checklist uses Java as an indirect reference for the speed comparison of AnyLogic

The following summarizes the features that are necessary in any programming package to be used in the redevelopment and further new development of the Tyche simulation engine. The features are presented in the following sections in a tabular format with "Yes/No" columns to ease in identifying the software package's fit to requirements. Additional comments may be added as necessary (for example, if the answer were no, but the package could accommodate the functionality in a different manner).

AnyLogic (Java): The Operating Environment

Over and above the programming language, there are a number of aspects of the operating environment that are significant to the effective development of a software project. The following should be considered (no particular order).

Requirement	Yes	No	Additional Comments
Native Code Compilation - Even in a multi-processing environment, processing speed is an important concern if we wish to "play nice" with other users of the hardware. Interpreted languages and languages that compile to p-code will not execute quickly enough.		no	While one may be able to compile bits of code, the main simulation would still run inside the software package (and cannot be compiled).
Error Handling - The existing simulation software contains extensive error handling code. Along with better pinpointing of problems, error handling also ensures that the simulation is able to recover from errors and continue processing with minimal loss of data and time. Runtime errors will happen. If the new system were to simply abort on an error (losing unsaved data) or only provided an end-state information dump, this would be unacceptable.	yes		try…catch
Symbolic Debugging (Debugging Environment) - Further to the "Error Handling" comments above, the complexity of the code dictates a need for a modern software-debugging environment (comparable to what has been provided in most mainstream development tools since the mid-1990s). Errors can be difficult to isolate and a proper debugging environment will assist in addressing problems promptly.	yes		e.g. Java Debugger (JDB), or IDE
Native Disk I/O Support - Though disk I/O is not currently performed during the iterative process in the current simulation, it is performed in the setting up of the simulation and in the reporting of results. Native disk I/O functions (as opposed to I/O through a third-party mechanism) performs significantly better. It is also possible that the approach to I/O may change as a consequence of multiprocessing. Results generated in each iteration might need to be written immediately. This would make native disk I/O even more important. Native disc I/O refers to reading and writing at the file, folder and drive levels.			
Native File I/O	yes		e.g. open(), close()
Native Folder I/O	yes		e.g. new File()
Native Drive I/O	yes		e.g. File.listRoots()
Command Line Support – The current simulation can be executed directly from a command line. The run's XML file name is passed through a command parameter. This functionality should still be available to the user.	yes		
Spawned Processing – Toward the end of the simulation run, the output file is compressed through the use of the WinZip Command Line Interface. This is accomplished through the dynamic spawning of a separate command process.	yes		e.g. Process.exec()

AnyLogic (Java): Data Handling

The simulation uses a rich and varied set of data structures to represent its configuration and the objects that are manipulated. During the running of the model, its state is determined by the data content of its objects. The new software development platform must provide data structuring mechanisms sufficient to define and store the data objects needed. The following sections outline those needs.

Requirement	Yes	No	Additional Comments
Record Definitions – In addition to the standard scalar variable types (string, integer, long, Boolean etc.), programmers must have a means of defining data structures that aggregate several different variables together so that data objects with multiple attributes can be created. At a minimum, a record definition mechanism must be provided. The current model uses more sophisticated data objects with properties and methods associated with them. Ideally, something similar to these "class" objects should also be supported.			
Scalar types that must be supported include:			
Byte	yes		byte
Boolean	yes		Boolean
Integer	yes		int
Long	yes		long
Single	yes		float
Double	yes		double
Date	yes		Date
String	yes		String
More complex structures include:			
Record Definitions	yes		Using classes
Complex Data Objects (Classes)	yes		classes
Dynamic Collections – Data collections are the heart and soul of the current simulation system. A collection mechanism provides for the dynamic grouping of data objects of common structure. Collections grow and shrink as the program proceeds. The collection mechanism must support adding and removing members dynamically. The number of items in a collection must be able to be determined at any time and the items in a collection must be able to be accessed and updated. Collection support is essential to any development tool to be selected.			
Require properties include:			
Adding/Removing items dynamically	yes		e.g. LinkedList, Hashset
"Count" property	yes		e.g. size()
Dynamic update of items	yes		e.g. change attributes of an item in LinkedList
Size limited only by available memory		no	e.g. the maximum index of items in a collection is equal to Integer.MAX_VALUE, i.e. 2147483647. It's however a log, you may hit the OutOfMemoryError sooner than the array size limit. Ref: http://stackoverflow.com/questions/3038392/java-do-arrays-have-a-maximum-size
Variable Scoping – The idea that a variable may have limitations to its scope has been pretty well accepted for quite some time. However, some languages (most notably COBOL) don't support variable scoping. Since nothing should be taken for granted with respect to what is supported in special purpose software development tools, variable scoping (localization) should be supported. A single global data pool would be too cumbersome.			
Variables should be able to be scoped as:			
Global	yes		e.g. public classes

Requirement	Yes	No	Additional Comments
Modular	yes		e.g. class, static variables
Procedural	yes		e.g. functions in classes

AnyLogic (Java): Processing Constructs

Requirement	Yes	No	Additional Comments
Modules – Modules allow code in large applications to be broken up logically into smaller and more manageable units. Some means of logically arranging program code must be supported in any development tool to be selected.	yes		classes
Procedures and Functions – Procedures and functions provide a means of dividing up processing logic into portions that can be called repeatedly. The programming language used by the new development tool must be procedural.	yes		Functions in classes
The following procedure types should be supported:			
Subroutines	yes		functions in classes
Functions	yes		Functions in classes
Both of these constructs should support parameter passing as follows:			
By reference		no	
By Value	yes		Ref: http://www.javaworld.com/javaworld/javaqa/2000-05/03-qa-0526-pass.html
Parameters should be either of the scalar types noted in the previous section and			
User defined record structures and			
Complex data objects	yes		Via classes
	yes		classes
Looping Structures – Most of the processing conducted in the model iterations involve looping structures of several types working within complex conditional processing. Both pre-test (FOR … NEXT, DO WHILE) and post-test (REPEAT … UNTIL) loop constructs are needed. Loops must also be able to be nested within each other.			
Looping structures should include the following:			
FOR … NEXT	yes		for
DO WHILE	yes		while…
REPEAT … UNTIL	yes		do…while
Conditional Constructs – As mentioned above, there is within the model a great amount of complex conditional processing. The programming language must support both IF… THEN … ELSE and CASE constructs. These structures must also be able to be nested within each other.			
The following conditional constructs must be supported:			
IF … THEN … ELSE	yes		if…else
CASE statements	yes		switch…case
In specifying conditions, complex logical constructs must be supported	yes		And, Or, Xor, ==, <, >, <=, >=, !=
Boolean functions should be able to be used within logical conditions	yes		And, Or, Xor, ==, <, >, <=, >-, !=

AnyLogic (Java): Miscellaneous Support

Requirement	Yes	No	Additional Comments
MS Excel Support – The current simulation provides some of its output in a Microsoft Excel workbook. The model directly supports outputting of information to Excel. This level of support isn't strictly needed as long as the necessary output can be formatted as either comma-delimited or tab-delimited text.	yes		Some free API packages available for this, e.g. Java Excel API, http://jexcelapi.sourceforge.net/
Random Number Generation – The current simulation uses random numbers extensively. The new development tool will need to be able to provide random numbers.	yes		e.g. Random()
The seed value for the random number generator will have to be adjustable programmatically.	yes		e.g. Random(long seed)

Simulation Development Software Functionality Checklist: SimEvents (Matlab)

Summary of SimEvents:

SimEvents and Simulink are based on Matlab. Therefore, the Matlab is used for the comparison between the checklist and SimEvents

The following summarizes the features that are necessary in any programming package to be used in the redevelopment and further new development of the Tyche simulation engine. The features are presented in the following sections in a tabular format with "Yes/No" columns to ease in identifying the software package's fit to requirements. Additional comments may be added as necessary (for example, if the answer were no, but the package could accommodate the functionality in a different manner).

Matlab: The Operating Environment

Over and above the programming language, there are a number of aspects of the operating environment that are significant to the effective development of a software project. The following should be considered (no particular order).

Requirement	Yes	No	Additional Comments
Native Code Compilation - Even in a multi-processing environment, processing speed is an important concern if we wish to "play nice" with other users of the hardware. Interpreted languages and languages that compile to p-code will not execute quickly enough.	no		While one may be able to compile bits of code, the main simulation would still run inside the software package (and cannot be compiled). Ref for a yes answer: http://www.mathworks.com/products/compiler/; http://www.mathworks.com/products/compiler/compiler_support.html

Requirement	Yes	No	Additional Comments
Error Handling - The existing simulation software contains extensive error handling code. Along with better pinpointing of problems, error handling also ensures that the simulation is able to recover from errors and continue processing with minimal loss of data and time. Runtime errors will happen. If the new system were to simply abort on an error (losing unsaved data) or only provided an end-state information dump, this would be unacceptable.	yes		Ref: http://www.mathworks.com/help/techdoc/matlab_prog/f3-38012.html
Symbolic Debugging (Debugging Environment) - Further to the "Error Handling" comments above, the complexity of the code dictates a need for a modern software-debugging environment (comparable to what has been provided in most mainstream development tools since the mid-1990s). Errors can be difficult to isolate and a proper debugging environment will assist in addressing problems promptly.	yes		Ref: http://www.mathworks.com/help/techdoc/matlab_prog/f3-38012.html; http://www.mathworks.com/help/techdoc/matlab_prog/f10-60570.html
Native Disk I/O Support - Though disk I/O is not currently performed during the iterative process in the current simulation, it is performed in the setting up of the simulation and in the reporting of results. Native disk I/O functions (as opposed to I/O through a third-party mechanism) performs significantly better. It is also possible that the approach to I/O may change as a consequence of multiprocessing. Results generated in each iteration might need to be written immediately. This would make native disk I/O even more important. Native disc I/O refers to reading and writing at the file, folder and drive levels.			
Native File I/O	yes		Ref: http://www.mathworks.com/support/tech-notes/1600/1602.html#general
Native Folder I/O	yes		Ref: http://www.mathworks.com/help/techdoc/ref/what.html; http://www.mathworks.com/help/techdoc/ref/mkdir.html
Native Drive I/O	yes		On Microsoft Windows platforms, to obtain a list of available drives, use the DOS net use command. In the Command Window, run [s,r] = dos('net use') # MATLAB returns the results to the character array r.
Command Line Support – The current simulation can be executed directly from a command line. The run's XML file name is passed through a command parameter. This functionality should still be available to the user.	yes		The matlab command prompt is available for start a matlab application. Ref: http://en.wikibooks.org/wiki/MATLAB_Programming/The_MATLAB_Command_Prompt
Spawned Processing – Toward the end of the simulation run, the output file is compressed through the use of the WinZip Command Line Interface. This is accomplished through the dynamic spawning of a separate command process.	yes		Matlab has its own compression functions: see http://www.mathworks.com/help/techdoc/ref/zip.html; see http://www.mathworks.com/matlabcentral/newsreader/view_thread/242949 for calling .exe files

Matlab: Data Handling

The simulation uses a rich and varied set of data structures to represent its configuration and the objects that are manipulated. During the running of the model, its state is determined by the data content of its objects. The new software development platform must provide data structuring mechanisms sufficient to define and store the data objects needed. The following sections outline those needs

Requirement	Yes	No	Additional Comments
Record Definitions – In addition to the standard scalar variable types (string, integer, long, Boolean etc.), programmers must have a means of defining data structures that aggregate several different variables together so that data objects with multiple attributes can be created. At a minimum, a record definition mechanism must be provided. The current model uses more sophisticated data objects with properties and methods associated with them. Ideally, something similar to these "class" objects should also be supported.			
Scalar types that must be supported include:			
Byte		no	Byte is from -127 to 128. Char is comparable, but char from 0 to 255 on 32-bit system, and 0 to 65535 on a 64-bit system
Boolean	yes		logical
Integer	yes		Numeric: int8, int16, int32, int64, uint8, uint16, uint32, uint64
Long	yes		Int 32, int64
Single	yes		Numeric - single
Double	yes		Numeric - double
Date	yes		e.g. Date String, Date Vector, Serial Date Number
String	yes		e.g. strings, char array
More complex structures include:			
Record Definitions	yes		e.g. struct
Complex Data Objects (Classes)	yes		e.g. class: Ref: http://www.mathworks.com/help/techdoc/matlab_prog/f2-38148.html ; http://www.mathworks.com/help/techdoc/matlab_oop/brh2rgw.html
Dynamic Collections – Data collections are the heart and soul of the current simulation system. A collection mechanism provides for the dynamic grouping of data objects of common structure. Collections grow and shrink as the program proceeds. The collection mechanism must support adding and removing members dynamically. The number of items in a collection must be able to be determined at any time and the items in a collection must be able to be accessed and updated. Collection support is essential to any development tool to be selected.	yes		Can have arrays of structs or objects
Require properties include:			
Adding/Removing items dynamically	yes	no	
"Count" property	no	no	Arrays do not have built in count property (use length())
Dynamic update of items	yes	no	
Size limited only by available memory	no	no	

Requirement	Yes	No	Additional Comments
Variable Scoping – The idea that a variable may have limitations to its scope has been pretty well accepted for quite some time. However, some languages (most notably COBOL) don't support variable scoping. Since nothing should be taken for granted with respect to what is supported in special purpose software development tools, variable scoping (localization) should be supported. A single global data pool would be too cumbersome. Variables should be able to be scoped as:			
Global	yes		e.g. global X Y Z. Ref: http://www.mathworks.com/help/techdoc/ref/global.html
Modular]	yes		e.g. packages, and a variety of workspaces : base workspace, global workspace, and function workspaces. Ref: http://msenux.redwoods.edu/Math4Textbook/Programming/VariableScope.pdf; http://www.dartmouth.edu/~rc/classes/matlab_prog/Matlab_Workspaces.html; http://www.mathworks.com/help/techdoc/matlab_oop/brfynt_-1.html
Procedural	yes		e.g. variables in function workspace. Ref: http://msenux.redwoods.edu/Math4Textbook/Programming/VariableScope.pdf

Matlab: Processing Constructs

Requirement	Yes	No	Additional Comments
Modules – Modules allow code in large applications to be broken up logically into smaller and more manageable units. Some means of logically arranging program code must be supported in any development tool to be selected.	yes		e.g. Ref: http://www.mathworks.com/help/techdoc/matlab_oop/brfynt_-1.html
Procedures and Functions – Procedures and functions provide a means of dividing up processing logic into portions that can be called repeatedly. The programming language used by the new development tool must be procedural. The following procedure types should be supported:			
Subroutines	yes		No subrountine concept found. However, a subfunction that follows a primary function in a file is comparable to a subroutine. It can be used by the primary function and other subfunctions in the file. Ref: http://www.mathworks.com/help/techdoc/matlab_prog/f4-70666.html
Functions	yes		e.g. primary function, nested functions, private functions, subfunctions.

Requirement	Yes	No	Additional Comments
Both of these constructs should support parameter passing as follows:			
By reference	yes		http://www.mathworks.com/support/solutions/en/data/1-15SO4/index.html?solution=1-15SO4
By Value	yes		This can be done for handle objects but not value objects.
Parameters should be either of the scalar types noted in the previous section and	yes		http://www.mathworks.com/support/solutions/en/data/1-15SO4/index.html?solution=1-15SO4
User defined record structures and			
Complex data objects	yes		e.g. struct. Ref: http://www.mathworks.com/help/techdoc/ref/struct.html
	yes		e.g. structs, cell arrays and user-defined classes. Ref: http://www.mathworks.com/help/techdoc/matlab_oop/bree1u6.html
Looping Structures – Most of the processing conducted in the model iterations involve looping structures of several types working within complex conditional processing. Both pre-test (FOR … NEXT, DO WHILE) and post-test (REPEAT … UNTIL) loop constructs are needed. Loops must also be able to be nested within each other.			
Looping structures should include the following:			
FOR … NEXT	yes		For…end
DO WHILE	yes		While…end
REPEAT … UNTIL		no	But may use alternative statements: e.g. t=true; while t t=some_condition; end
Conditional Constructs – As mentioned above, there is within the model a great amount of complex conditional processing. The programming language must support both IF… THEN … ELSE and CASE constructs. These structures must also be able to be nested within each other.			
The following conditional constructs must be supported:			
IF … THEN … ELSE	yes		e.g. if…elseif…else…end
CASE statements	yes		e.g. switch…case…otherwise…end
In specifying conditions, complex logical constructs must be supported	yes		e.g. AND, OR, NOT, &&, ||, &, |, ~, <, >, <=, >=, ==, ~=
Boolean functions should be able to be used within logical conditions	yes		e.g. and(), or(), not(), xor(), all, any, isa, …

Matlab: Miscellaneous Support

Requirement	Yes	No	Additional Comments

Requirement	Yes	No	Additional Comments
MS Excel Support – The current simulation provides some of its output in a Microsoft Excel workbook. The model directly supports outputting of information to Excel. This level of support isn't strictly needed as long as the necessary output can be formatted as either comma-delimited or tab-delimited text.	yes		Matlab can definitely call Excel. Ref: http://www.mathworks.com/help/techdoc/ref/xlsread.html
Random Number Generation – The current simulation uses random numbers extensively. The new development tool will need to be able to provide random numbers.	yes		Ref: http://www.mathworks.com/help/techdoc/math/bruv81b.html; http://www.mathworks.com/help/techdoc/ref/xlswrite.html
The seed value for the random number generator will have to be adjustable programmatically.	yes		Ref: http://www.mathworks.com/help/techdoc/ref/randstream.randstream.html

Simulation Development Software Functionality Checklist: Simio (Visual C#)

Summary of Simio:

1. Simio is an integrated simulation tool with a graphical user interface. It is at the higher level than common programming languages. In many cases, users do not need to write programming code to generate simulation models with the Simio GUI. However, users can use .NET languages to extend Simio for complex models.
2. Simio's GUI cannot provide the functionality in the checklist directly, but it is indirectly comparable to Visual C# because it is implemented in C# and can be extended in C# or .NET languages. Ref: http://www.informs-sim.org/wsc10papers/001.pdf; http://www.simio.com/about-simio/why-simio/simio-is-the-future-of-simulation-software-growing-with-you.html

The following summarizes the features that are necessary in any programming package to be used in the redevelopment and further new development of the Tyche simulation engine. The features are presented in the following sections in a tabular format with "Yes/No" columns to ease in identifying the software package's fit to requirements. Additional comments may be added as necessary (for example, if the answer were no, but the package could accommodate the functionality in a different manner).

Simio (VC#): The Operating Environment

Over and above the programming language, there are a number of aspects of the operating environment that are significant to the effective development of a software project. The following should be considered (no particular order).

Requirement	Yes	No	Additional Comments
Native Code Compilation - Even in a multi-processing environment, processing speed is an important concern if we wish to "play nice" with other users of the hardware. Interpreted languages		no	While one may be able to compile bits of code, the main simulation would still run inside the software package

Requirement	Yes	No	Additional Comments
and languages that compile to p-code will not execute quickly enough.			(and cannot be compiled).
Error Handling - The existing simulation software contains extensive error handling code. Along with better pinpointing of problems, error handling also ensures that the simulation is able to recover from errors and continue processing with minimal loss of data and time. Runtime errors will happen. If the new system were to simply abort on an error (losing unsaved data) or only provided an end-state information dump, this would be unacceptable.	yes		Try catch exception
Symbolic Debugging (Debugging Environment) - Further to the "Error Handling" comments above, the complexity of the code dictates a need for a modern software-debugging environment (comparable to what has been provided in most mainstream development tools since the mid-1990s). Errors can be difficult to isolate and a proper debugging environment will assist in addressing problems promptly.	yes		
Native Disk I/O Support - Though disk I/O is not currently performed during the iterative process in the current simulation, it is performed in the setting up of the simulation and in the reporting of results. Native disk I/O functions (as opposed to I/O through a third-party mechanism) performs significantly better. It is also possible that the approach to I/O may change as a consequence of multiprocessing. Results generated in each iteration might need to be written immediately. This would make native disk I/O even more important. Native disc I/O refers to reading and writing at the file, folder and drive levels.			
Native File I/O	yes		e.g. open(), read(), write()
Native Folder I/O	yes		e.g. createDirectory(), open(), read(), write()
Native Drive I/O	yes		e.g. GetLogicDrives()
Command Line Support – The current simulation can be executed directly from a command line. The run's XML file name is passed through a command parameter. This functionality should still be available to the user.	yes		
Spawned Processing – Toward the end of the simulation run, the output file is compressed through the use of the WinZip Command Line Interface. This is accomplished through the dynamic spawning of a separate command process.	yes		e.g. System.Diagnostics.Process(0

Simio (VC#): Data Handling

The simulation uses a rich and varied set of data structures to represent its configuration and the objects that are manipulated. During the running of the model, its state is determined by the data content of its objects. The new software development platform must provide data structuring mechanisms sufficient to define and store the data objects needed. The following sections outline those needs

Requirement	Yes	No	Additional Comments
Record Definitions – In addition to the standard scalar variable types (string, integer, long, Boolean etc.), programmers must have a means of defining data structures that aggregate several different variables together so that data objects with multiple attributes can be created. At a minimum, a record definition mechanism must be provided. The current model uses more sophisticated data objects with properties and methods associated with them. Ideally, something similar to these "class" objects should also be supported.			
Scalar types that must be supported include:			
Byte	yes		byte
Boolean	yes		bool
Integer	yes		int
Long	yes		long
Single	yes		float
Double	yes		double
Date	yes		DateTime
String	yes		string
More complex structures include:			
Record Definitions	yes		struct
Complex Data Objects (Classes)	yes		class
Dynamic Collections – Data collections are the heart and soul of the current simulation system. A collection mechanism provides for the dynamic grouping of data objects of common structure. Collections grow and shrink as the program proceeds. The collection mechanism must support adding and removing members dynamically. The number of items in a collection must be able to be determined at any time and the items in a collection must be able to be accessed and updated. Collection support is essential to any development tool to be selected.			
Require properties include:			
Adding/Removing items dynamically	yes		e.g. Collection, list, linkedList, hashtable
"Count" property	yes		e.g. Count()
Dynamic update of items	yes		e.g. changing attributes of an item in a List
Size limited only by available memory		no	The maximum number of elements in a collection is Int32.MaxValue – just over two billion. In practice, you'll most likely run out of allocable memory before you reach those limits, especially if you're running on a 32-bit system. Ref: http://stackoverflow.com/questions/3906891/what-is-the-max-limit-of-data-into-liststring-in-c

Requirement	Yes	No	Additional Comments
Variable Scoping – The idea that a variable may have limitations to its scope has been pretty well accepted for quite some time. However, some languages (most notably COBOL) don't support variable scoping. Since nothing should be taken for granted with respect to what is supported in special purpose software development tools, variable scoping (localization) should be supported. A single global data pool would be too cumbersome. Variables should be able to be scoped as:			
Global	yes		e.g. public classes, static variables
Modular	yes		e.g. classes
Procedural	yes		e.g. functions in classes

Simio (VC#): Processing Constructs

Requirement	Yes	No	Additional Comments
Modules – Modules allow code in large applications to be broken up logically into smaller and more manageable units. Some means of logically arranging program code must be supported in any development tool to be selected.	yes		classes
Procedures and Functions – Procedures and functions provide a means of dividing up processing logic into portions that can be called repeatedly. The programming language used by the new development tool must be procedural. The following procedure types should be supported:			
Subroutines	yes		e.g. calling functions of an object
Functions	yes		e.g. functions in classes
Both of these constructs should support parameter passing as follows:			
By reference	yes		Ref: http://msdn.microsoft.com/en-us/library/0f66670z%28v=vs.71%29.aspx
By Value	yes		Ref: http://msdn.microsoft.com/en-us/library/0f66670z%28v=vs.71%29.aspx
Parameters should be either of the scalar types noted in the previous section and			
User defined record structures and			
Complex data objects	yes		e.g. struct
	yes		e.g. objects
Looping Structures – Most of the processing conducted in the model iterations involve looping structures of several types working within complex conditional processing. Both pre-test (FOR … NEXT, DO WHILE) and post-test (REPEAT … UNTIL) loop constructs are needed. Loops must also be able to be nested within each other. Looping structures should include the following:			
FOR … NEXT	yes		for, foreach
DO WHILE	yes		while; do…while
REPEAT … UNTIL	yes		do…while

Requirement	Yes	No	Additional Comments
Conditional Constructs – As mentioned above, there is within the model a great amount of complex conditional processing. The programming language must support both IF… THEN … ELSE and CASE constructs. These structures must also be able to be nested within each other.			
The following conditional constructs must be supported:			
IF … THEN … ELSE	yes		if…else
CASE statements	yes		switch…case…default
In specifying conditions, complex logical constructs must be supported	yes		AND, OR, XOR, NOT, ==, <, >, <=, >=, !=, <<, shifted
Boolean functions should be able to be used within logical conditions	yes		AND, OR, XOR, NOT, ==, <, >, <=, >=, !=, <<

Simio (VC#): Miscellaneous Support

Requirement	Yes	No	Additional Comments
MS Excel Support – The current simulation provides some of its output in a Microsoft Excel workbook. The model directly supports outputting of information to Excel. This level of support isn't strictly needed as long as the necessary output can be formatted as either comma-delimited or tab-delimited text.	yes		Ref. http://support.microsoft.com/kb/302084
Random Number Generation – The current simulation uses random numbers extensively. The new development tool will need to be able to provide random numbers.	yes		e.g. Random class
The seed value for the random number generator will have to be adjustable programmatically.	yes		e.g. Random (int seed)

Simulation Development Software Functionality Checklist: SimPy (Python)

Summary of SimPy:

SimPy is an add on package to Python. Therefore, it functionality directly depends on the functionality of Python.

The following summarizes the features that are necessary in any programming package to be used in the redevelopment and further new development of the Tyche simulation engine. The features are presented in the following sections in a tabular format with "Yes/No" columns to ease in identifying the software package's fit to requirements. Additional comments may be added as necessary (for example, if the answer were no, but the package could accommodate the functionality in a different manner).

SimPy (Python): The Operating Environment

Over and above the programming language, there are a number of aspects of the operating environment that are significant to the effective development of a software project. The following should be considered (no particular order).

Requirement	Yes	No	Additional Comments
Native Code Compilation - Even in a multi-processing environment, processing speed is an important concern if we wish to "play nice" with other users of the hardware. Interpreted languages and languages that compile to p-code will not execute quickly enough.		no	Interpreted
Error Handling - The existing simulation software contains extensive error handling code. Along with better pinpointing of problems, error handling also ensures that the simulation is able to recover from errors and continue processing with minimal loss of data and time. Runtime errors will happen. If the new system were to simply abort on an error (losing unsaved data) or only provided an end-state information dump, this would be unacceptable.	yes		try…except
Symbolic Debugging (Debugging Environment) - Further to the "Error Handling" comments above, the complexity of the code dictates a need for a modern software-debugging environment (comparable to what has been provided in most mainstream development tools since the mid-1990s). Errors can be difficult to isolate and a proper debugging environment will assist in addressing problems promptly.	yes		Some IDE can provide such functionality, e.g. WingIDE
Native Disk I/O Support - Though disk I/O is not currently performed during the iterative process in the current simulation, it is performed in the setting up of the simulation and in the reporting of results. Native disk I/O functions (as opposed to I/O through a third-party mechanism) performs significantly better. It is also possible that the approach to I/O may change as a consequence of multiprocessing. Results generated in each iteration might need to be written immediately. This would make native disk I/O even more important. Native disc I/O refers to reading and writing at the file, folder and drive levels.			
Native File I/O	yes		e.g. open(), read(), write()
Native Folder I/O	yes		e.g. os.path.exists(dir), os.makedirs(dir)
Native Drive I/O	yes		e.g. Ref: http://www.daniweb.com/software-development/python/threads/168176; http://www.mail-archive.com/python-list@python.org/msg194250.html
Command Line Support – The current simulation can be executed directly from a command line. The run's XML file name is passed through a command parameter. This functionality should still be available to the user.	yes		
Spawned Processing – Toward the end of the simulation run, the output file is compressed through the use of the WinZip Command Line Interface. This is accomplished through the dynamic spawning of a separate command process.	yes		e.g. from superprocess import call; call([cmdString, parameters]). Ref: http://stackoverflow.com/questions/89228/how-to-call-external-command-in-python

SimPy (Python): Data Handling

The simulation uses a rich and varied set of data structures to represent its configuration and the objects that are manipulated. During the running of the model, its state is determined by the data content of its objects. The new software development platform must provide data structuring mechanisms sufficient to define and store the data objects needed. The following sections outline those needs

Requirement	Yes	No	Additional Comments
Record Definitions – In addition to the standard scalar variable types (string, integer, long, Boolean etc.), programmers must have a means of defining data structures that aggregate several different variables together so that data objects with multiple attributes can be created. At a minimum, a record definition mechanism must be provided. The current model uses more sophisticated data objects with properties and methods associated with them. Ideally, something similar to these "class" objects should also be supported.			
Scalar types that must be supported include:			
Byte		no	
Boolean	yes		
Integer	yes		integer
Long	yes		long integer
Single		no	
Double	yes		float
Date	yes		date
String	yes		Character string
More complex structures include:			
Record Definitions	yes		e.g. dictionary, list, set
Complex Data Objects (Classes)	yes		classes
Dynamic Collections – Data collections are the heart and soul of the current simulation system. A collection mechanism provides for the dynamic grouping of data objects of common structure. Collections grow and shrink as the program proceeds. The collection mechanism must support adding and removing members dynamically. The number of items in a collection must be able to be determined at any time and the items in a collection must be able to be accessed and updated. Collection support is essential to any development tool to be selected.			
Require properties include:			
Adding/Removing items dynamically	yes		e.g. list, set, insert(), remove()
"Count" property	yes		count
Dynamic update of items	yes		e.g. change attributes in an item in a list
Size limited only by available memory		no	Maximum size of a list PY_SSIZE_T_MAX/sizeof**PyObject*). It is 536870912 on a 32-bit machine. Ref: http://stackoverflow.com/questions/855191/how-big-can-a-python-array-get

Requirement	Yes	No	Additional Comments
Variable Scoping – The idea that a variable may have limitations to its scope has been pretty well accepted for quite some time. However, some languages (most notably COBOL) don't support variable scoping. Since nothing should be taken for granted with respect to what is supported in special purpose software development tools, variable scoping (localization) should be supported. A single global data pool would be too cumbersome. Variables should be able to be scoped as:			
Global	yes		Declare a variable as global
Modular	yes		class
Procedural	yes		function

SimPy (Python): Processing Constructs

Requirement	Yes	No	Additional Comments
Modules – Modules allow code in large applications to be broken up logically into smaller and more manageable units. Some means of logically arranging program code must be supported in any development tool to be selected.	yes		modules
Procedures and Functions – Procedures and functions provide a means of dividing up processing logic into portions that can be called repeatedly. The programming language used by the new development tool must be procedural. The following procedure types should be supported:	yes		Functions in classes
Subroutines	yes		functions
Functions	yes		functions
Both of these constructs should support parameter passing as follows:			
By reference	yes		Note: mutable variables are passed by reference. Ref: http://bogdan.org.ua/2008/02/11/python-passing-by-value-vs-passing-by-reference.html
By Value	yes		Note: immutable (string, numbers and tuples) variables are passed by values. Ref: http://bogdan.org.ua/2008/02/11/python-passing-by-value-vs-passing-by-reference.html; **http://stackoverflow.com/**questions/986006/python-how-do-i-pass-a-variable-by-reference
Parameters should be either of the scalar types noted in the previous section and User defined record structures and Complex data objects			
	yes		e.g. list
	yes		classes

Requirement	Yes	No	Additional Comments
Looping Structures – Most of the processing conducted in the model iterations involve looping structures of several types working within complex conditional processing. Both pre-test (FOR … NEXT, DO WHILE) and post-test (REPEAT … UNTIL) loop constructs are needed. Loops must also be able to be nested within each other. Looping structures should include the following:			
FOR … NEXT	yes		for…in
DO WHILE	yes		while
REPEAT … UNTIL	yes		do…while
Conditional Constructs – As mentioned above, there is within the model a great amount of complex conditional processing. The programming language must support both IF… THEN … ELSE and CASE constructs. These structures must also be able to be nested within each other. The following conditional constructs must be supported:			
IF … THEN … ELSE	yes		If…else
CASE statements	yes		switch…case…default
In specifying conditions, complex logical constructs must be supported	yes		e.g. and, or, xor, complement, shift, ==, !=, <, >, <=, >=
Boolean functions should be able to be used within logical conditions	yes		e.g. and, or, xor, complement, shift

SimPy (Python): Miscellaneous Support

Requirement	Yes	No	Additional Comments
MS Excel Support – The current simulation provides some of its output in a Microsoft Excel workbook. The model directly supports outputting of information to Excel. This level of support isn't strictly needed as long as the necessary output can be formatted as either comma-delimited or tab-delimited text.	yes		http://www.python-excel.org/ Did not find direct way, but there is some method for that, e.g. http://oreilly.com/catalog/pythonwin32/chapter/ch12.html#49339;
Random Number Generation – The current simulation uses random numbers extensively. The new development tool will need to be able to provide random numbers.	yes		Random class
The seed value for the random number generator will have to be adjustable programmatically.	yes		random.seed(seedNum)

Simulation Development Software Functionality Checklist: Simul8 (Visual Logic)

Summary of Simul8:

Simul8 provides an internal programming language, called Visual Logic, for users to develop complex models. The results of checklist for Simul8 are based on the functionality of Visual Logic.

The following summarizes the features that are necessary in any programming package to be used in the redevelopment and further new development of the Tyche simulation engine. The features are presented in the following sections in a tabular format with "Yes/No" columns to ease in identifying the software package's fit to requirements. Additional comments may be added as necessary (for example, if the answer were no, but the package could accommodate the functionality in a different manner).

Visual Logic: The Operating Environment

Over and above the programming language, there are a number of aspects of the operating environment that are significant to the effective development of a software project. The following should be considered (no particular order).

Requirement	Yes	No	Additional Comments
Native Code Compilation - Even in a multi-processing environment, processing speed is an important concern if we wish to "play nice" with other users of the hardware. Interpreted languages and languages that compile to p-code will not execute quickly enough.		?	Did not find information about how to handle Visual Logic code within Simul8. While one may be able to compile bits of code, the main simulation would still run inside the software package (and cannot be compiled).
Error Handling - The existing simulation software contains extensive error handling code. Along with better pinpointing of problems, error handling also ensures that the simulation is able to recover from errors and continue processing with minimal loss of data and time. Runtime errors will happen. If the new system were to simply abort on an error (losing unsaved data) or only provided an end-state information dump, this would be unacceptable.		?	Did not find the means for error handling in Visual Logic.
Symbolic Debugging (Debugging Environment) - Further to the "Error Handling" comments above, the complexity of the code dictates a need for a modern software-debugging environment (comparable to what has been provided in most mainstream development tools since the mid-1990s). Errors can be difficult to isolate and a proper debugging environment will assist in addressing problems promptly.	yes		Ref: http://www.simul8.com/products/faq.htm#debugger
Native Disk I/O Support - Though disk I/O is not currently performed during the iterative process in the current simulation, it is performed in the setting up of the simulation and in the reporting of results. Native disk I/O functions (as opposed to I/O through a third-party mechanism) performs significantly better. It is also possible that the approach to I/O may change as a consequence of multiprocessing. Results generated in each iteration might need to be written immediately. This would make native disk I/O even more important. Native disc I/O refers to reading and writing at the file, folder and drive levels.			
Native File I/O	yes		Ref: "Direct File Read/Write in Visual Logic" in http://www.hallogram.com/simul/index.html
Native Folder I/O		?	Did not find information to create a folder.
Native Drive I/O		?	Did not find information to list hard drives.
Command Line Support – The current simulation can be executed directly from a command line. The run's XML file name is passed through a command parameter. This functionality should still be available to the user.		?	Did not find information
Spawned Processing – Toward the end of the simulation run, the output file is compressed through the use of the WinZip Command Line Interface. This is accomplished through the dynamic spawning of a separate command process.		?	Did not find information

Visual Logic: Data Handling

The simulation uses a rich and varied set of data structures to represent its configuration and the objects that are manipulated. During the running of the model, its state is determined by the data content of its objects. The new software development platform must provide data structuring mechanisms sufficient to define and store the data objects needed. The following sections outline those needs

Requirement	Yes	No	Additional Comments
Record Definitions – In addition to the standard scalar variable types (string, integer, long, Boolean etc.), programmers must have a means of defining data structures that aggregate several different variables together so that data objects with multiple attributes can be created. At a minimum, a record definition mechanism must be provided. The current model uses more sophisticated data objects with properties and methods associated with them. Ideally, something similar to these "class" objects should also be supported.			
Scalar types that must be supported include:			
Byte……		no	Did not find the Byte data type.
Boolean……		no	Did not find Boolean.
Integer……	yes		e.g., the "number" type in local variable types: number, text, sheet, simulation object
Long……....	yes		e.g., the "number" type in local variable types: number, text, sheet, simulation object
Single……..	yes		e.g., the "number" type in local variable types: number, text, sheet, simulation object
Double……	yes		e.g., the "number" type in local variable types: number, text, sheet, simulation object
Date……..	yes		e.g. math functions: DAY, HOUR, MINIUTE & WEEK
String…….	yes		e.g. the "text" type in local variable types: number, text, sheet, simulation object
More complex structures include:			
Record Definitions	yes		e.g. the "simulation object" type, "sheet" type
Complex Data Objects (Classes)	yes		e.g. the "simulation object" type
Dynamic Collections – Data collections are the heart and soul of the current simulation system. A collection mechanism provides for the dynamic grouping of data objects of common structure. Collections grow and shrink as the program proceeds. The collection mechanism must support adding and removing members dynamically. The number of items in a collection must be able to be determined at any time and the items in a collection must be able to be accessed and updated. Collection support is essential to any development tool to be selected.			http://www.simul8.com/support/help/doku.php?id=tips:excel_interface
Require properties include:			
Adding/Removing items dynamically	yes		Ref: Spreadsheets can be referenced and set from Visual Logic and all simulation objects. http://www.simul8.com/products/features/data.htm

Requirement	Yes	No	Additional Comments
"Count" property	yes		Ref: Spreadsheets can be referenced and set from Visual Logic and all simulation objects. http://www.simul8.com/products/features/data.htm
Dynamic update of items	yes		Ref: Spreadsheets can be referenced and set from Visual Logic and all simulation objects. http://www.simul8.com/products/features/data.htm
Size limited only by available memory		?	Did not find information
Variable Scoping – The idea that a variable may have limitations to its scope has been pretty well accepted for quite some time. However, some languages (most notably COBOL) don't support variable scoping. Since nothing should be taken for granted with respect to what is supported in special purpose software development tools, variable scoping (localization) should be supported. A single global data pool would be too cumbersome. Variables should be able to be scoped as:			
Global	yes		Ref: "any variable you declare in the Information Store is global. "http://www.simul8.com/support/help/doku.php?id=features:visual_logic:localvars
Modular		?	Ref: Visual Logic libraries can contain multiple VL sections, but did not find information about variable scope in libraries. "http://www.simul8.com/support/help/doku.php?id=features:visual_logic:libraries
Procedural	yes		Ref: http://www.simul8.com/support/help/doku.php?id=features:visual_logic:localvars; http://simul8.com/support/help/doku.php?id=features:visual_logic:functions

Visual Logic: Processing Constructs

Requirement	Yes	No	Additional Comments
Modules – Modules allow code in large applications to be broken up logically into smaller and more manageable units. Some means of logically arranging program code must be supported in any development tool to be selected.	yes		Ref: http://www.simul8.com/support/help/doku.php?id=features:visual_logic:libraries
Procedures and Functions – Procedures and functions provide a means of dividing up processing logic into portions that can be called repeatedly. The programming language used by the new development tool must be procedural. The following procedure types should be supported:			
		?	Did not find information

Requirement	Yes	No	Additional Comments
Subroutines Functions	yes		Ref: http://www.simul8.com/products/features/vl.htm#functions
Both of these constructs should support parameter passing as follows:		?	Did not find information
By reference		?	Did not find information
By Value			
Parameters should be either of the scalar types noted in the previous section and	yes		e.g. sheets, simulation objects
User defined record structures and Complex data objects	yes		e.g. simulation objects
Looping Structures – Most of the processing conducted in the model iterations involve looping structures of several types working within complex conditional processing. Both pre-test (FOR … NEXT, DO WHILE) and post-test (REPEAT … UNTIL) loop constructs are needed. Loops must also be able to be nested within each other. Looping structures should include the following:			
FOR … NEXT	yes		e.g. loop…count
DO WHILE	yes		e.g. while
REPEAT … UNTIL	yes		e.g. until
Conditional Constructs – As mentioned above, there is within the model a great amount of complex conditional processing. The programming language must support both IF… THEN … ELSE and CASE constructs. These structures must also be able to be nested within each other. The following conditional constructs must be supported:			
IF … THEN … ELSE	yes		e.g. if…elseif…else
CASE statements		no	Did not find information
In specifying conditions, complex logical constructs must be supported	yes		e.g. <, >, =, http://www.simul8.com/support/help/doku.php?id=features:visual_logic:commands:complexif
Boolean functions should be able to be used within logical conditions	yes		e.g. &, |; http://www.simul8.com/support/help/doku.php?id=features:visual_logic:commands:complexif

Visual Logic: Miscellaneous Support

Requirement	Yes	No	Additional Comments
MS Excel Support – The current simulation provides some of its output in a Microsoft Excel workbook. The model directly supports outputting of information to Excel. This level of support isn't strictly needed as long as the necessary output can be formatted as either comma-delimited or tab-delimited text.	yes		Ref: http://www.simul8.com/support/newsletter/SIMUL8_User/ExcelInterfaces.htm
Random Number Generation – The current simulation uses random numbers extensively. The new development tool will need to be able to provide random numbers.	yes		Ref: http://www.simul8.com/products/features/results.htm
The seed value for the random number generator will have to be adjustable programmatically.	yes		Ref: http://www.simul8.com/support/help/doku.php?id=features:visual_logic:commands:set_vl_random_number_seed

List of Symbols/Abbreviations/Acronyms/Initialisms

API	Application Programming Interface
CF	Canadian Forces
CLI	Command Line Interface
COM	Component Object Mode
CORA	Center for Operational Research and Analysis
DES	Discrete Event Simulation
DND	Department of National Defence
DRDC	Defence Research and Development Canada
FCFS	First Come First Serve
FSM	Finite State Machine
GUI	Graphical User Interface
HLA	High Level Architecture
HPC	High-Performance Computing
I/O	Input / Output
iSCSI	Internet Small Computer System Interface
LP	Logic Process
MD	Model Development, or Model Development Tool, or Model Development GUI
MPI	Message-Passing Interface
M&S	Modelling and Simulation
MS	Microsoft
NATO	North Atlantic Treaty Organization
OpenMP	Open Multi-Processing
OpSched	Operational Schedule
PCAN	Partitioning, Communication, Agglomeration, and Mapping
PDES	Parallel Discrete Event Simulation
PDSS	Parallel and Distributed Simulation System
RDMA	Remote Direct Memory Access
SE	Simulation Engine
SM	Simulation Management, or Simulation Management Tool, or Simulation Management GUI
SNFL	Sanding Naval Forces Atlantic (NATO)

TMD	Tyche Model Development Tool
TPM	Tyche Parallelization Management Tool
TSE	Tyche Simulation Engine
TSM	Tyche Simulation Management Tool
TYI	Tyche input file
TYO	Tyche output file
UI	User Interface
VC++	Microsoft Visual C++
VC#	Microsoft Visual C#
VB	Microsoft Visual Basic
VBScripts	Visual Basic Scripts
XML	Extensible Markup Language

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.) CAE Professional Services 300-1135 Innovation Drive Ottawa, ON K2K 3G7 CANADA	2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.) UNCLASSIFIED (NON-CONTROLLED GOODS) DMC A REVIEW: December 2012

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.)

Analysis and Recommendations of the Tyche Simulation System

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

Ruibiao Jaff Guo; Jeremy Brooks

5. DATE OF PUBLICATION (Month and year of publication of document.) APRIL 2012	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.) 124	6b. NO. OF REFS (Total cited in document.) 118

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

Contract Report

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

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

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.) 11ic	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.) W7714-4500825828
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.) Contractor's Document Number: CAE PS #5160-016 Version 01 CAE PS #5160-016 Version 01	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.) DRDC CORA CR 2012-081

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 : The objective of this project was to analyze the current Tyche simulation system, identify the problems that affect simulation execution time, and make recommendations to speed up Tyche simulation execution.

The analysis of the Tyche system consisted of identifying physical, conceptual and dynamic entities/components as well as their relationships, and extracting function flowcharts in the simulation process to help understand the system and diagnose problems.

Based on the results of the Tyche system analysis, two kinds of problems were identified that affect simulation execution time including the problem of single-thread execution and the issues of redundant or repeated operations in simulation iterations.

Three categories of recommendations are proposed for speeding up simulation execution and future Tyche development, comprising a parallel system architecture, a group of approaches for optimizing the design of the simulation engine, and a set of alternative programming environments for future Tyche development.

Le projet avait pour but d'analyser le système de simulation Tyche actuel, de cerner les problèmes qui nuisent à la vitesse d'exécution et de formuler des recommandations en vue d'accélérer l'exécution de simulations Tyche.

L'analyse du système Tyche comportait l'identification des composants et des entités physiques, conceptuelles et dynamiques, ainsi que des relations entre ceux-ci, et l'extraction d'organigrammes des fonctions du processus de simulation pour permettre de mieux comprendre le système et de diagnostiquer les problèmes.

Les résultats de l'analyse du système Tyche ont permis de cerner deux types de problèmes qui nuisent à la vitesse d'exécution de simulations : utilisation d'un seul fil d'exécution et opérations redondantes ou répétées au cours des itérations de simulation.

Nous proposons trois catégories de recommandations pour accélérer les simulations et guider le développement futur : une architecture système parallèle, un ensemble de démarches pour optimiser la conception du moteur de simulation et un ensemble d'autres environnements de programmation pour le développement futur de Tyche.

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

Tyche Simulation, Discrete event simulation, parallel discrete event simulation, analysis, recommendation, optimization, programming environment, military force structures, Windows HPC Server 2008 R2

Defence R&D Canada

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

R & D pour la défense Canada

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

DEFENCE DÉFENSE

www.drdc-rddc.gc.ca