Exploring Composability within Simulations-as-a-Service Paradigm

by

Chesley Montague and Robin Sexton

An Applied Project Presented in Partial Fulfillment
of the Requirements for the Degree
Masters of Engineering

Approved December 2012 by the
Graduate Supervisory Committee:

Hessam Sarjoughian, Chair

ARIZONA STATE UNIVERSITY

December 2012

ABSTRACT

This capstone project examined in detail some of the aspects of Simulation-As-A-Service (SimAAS) concept. The project included research and development in four of the five areas of a Software-as-a-Service (SaaS) system. The four areas are *Publishing*, *Discovery*, *Composition*, and *Deployment*. Monitoring and policy enforcement will not be examined in this project.

This project produced three products: a web interface Composer, a simple simulation, and a discussion of the information required for the Publish/Discovery process.

This project focused on the composition of basic and complex models into a system where a tenant controls what models are present in simulation built to the tenants needs. This project demonstrated the ability and the actions required in a multi tenant environment for this process. The investigation has shown that there needs to be further research into the areas of data management used to store the information on model composition. There are other component based systems that have done research into alternate methods such as SOSA, FCINT simulation, or web based services to describe the Simulation-as-a-Service paradigm that are worth further study. In addition more work needs to be done in IAAS, and PAAS domains to work requirements needed for large scale distributed simulation. Lastly, security will become increasingly important if the architecture is used among tenants that do not want to share information with other tenants which leads to more research being done into encryption needed for all aspects of the Simulation-as-a-Service architecture and framework.

ACKNOWLEDGMENTS

First, we would like to express the deepest appreciation to our committee chair and advisor, Dr. Hessam Sarjoughian. His guidance and advice was always poignant and given in a friendly manner. We especially appreciate his patients with our long dissentience relationship. Without his guidance and persistent help this project and our career as students at ASU would not have been possible.

We also recognize the help of the Fires Battle Lab, specifically Lt Col (Ret) Chris Niederhauser and Jeffrey Milam, for providing resources and allowing time in an already challenging schedule to work on this project.

Finally we thank Arizona State University and there online graduate program for allowing non-traditional students to further their education. The staff and administration have always been friendly and responsive.

TABLE OF CONTENTS

LIST OF FIGURES

Objective

The intent of this capstone project is to examine in detail some of the aspects of Simulation-As-A-Service (SimAAS) concept. The project included research and development in four of the five areas of a Software-as-a-Service (SaaS) system. The four areas are *Publishing*, *Discovery*, *Composition*, and *Deployment*. Monitoring and policy enforcement will not be examined in this project.

The focus of this project is from a simulation point of view. The completed system is a functional simulation (as opposed to the simulation and testing component of a traditional SaaS system).

This project will produce three products: a web interface Composer, a simple simulation, and a discussion of the information required for the Publish/Discovery process.

Motivation

Since the early 90s the US Army uses distributed simulations for experimentation, analysis, and training.

Experimentation and analysis with distributed simulation consists of federations of disparate simulations with each branch of the army being represented by their specialized federates and subject matter experts as role players, analyst and technical staff. Role players are typically organized into Blue cells and Red cells. The analyst and technical staff are included as a White cell. Depending on a specific problem under study the number of federates involved can be over thirty and the number of personal involved reach over 500. The testing, integration, and execution cycle can be as long as 9 months.

In the training domain distributed simulation is used for command post staff training and unit mission rehearsal. Staff training typically uses the simulation environment to produce a realistic scenario. Emphasis is placed on stimulating the real go-to-war systems accurately. The soldier operating the tactical box is presented with information that represents real scenarios in order to train the command post staff in proper decision making and reactions. The scale of these exercise are typically smaller than an experiment. The federation may have only a few simulations and the participants under 100. Complexity is added by the inclusion of command and control systems that are in use by the active military.

Problems are inherent in this environment because these federations are composed of simulations built and managed by many different software development teams with various skills and development practices. Integration and testing are costly and may not ensure a stable execution environment. The simulation environment may not provide the

fidelity or capability required by a study director. The study director is stuck with a limited number of options for the scale and composition of the federation.

Simulation-as-a-Service may provide a solution to some of these problems. Before resources are applied to a full scale investigation and test of a SimAAS environment preliminary investigation will need to be preformed and insights provided to leadership.

This project will provide insight to a future Fires Battle white paper discussing the use of Simulation-as-a-Service as a potential component of Cloud-Based Training-As-a-Service (CBTAS). If successful, this project will lead to future work within the US Army on Simulations-as-a-service.

Scope of Study

The goal of this project is to construct a Simulation-as-a-Service environment of sufficient scale in order to examine the requirements and intricacies of the *Publish* and *Discovery* phases within the Software as a Service (SaaS) paradigm. Publishing and Discovery define the interactions between different models. A firm understanding of the links between models is one of the first steps in developing interface specifications for engineers to work with.

The complete environment for a Simulation-as-a-Service (SimAAS) system using a SaaS paradigm would consist of repository of models on multiple sites publishing to a Composer the information necessary to be included within a simulation environment. The Composer would discover the models available and present the proper information to a user allowing a rich and variable simulation capability.

In order to determine the requirements for the publish/discover relationship between models and the user an end-to-end simulation system was constructed. This system consists of a number of models, a composer, and a simulation engine.

Component classification is used in some prescribed research to show how these components handle time. This type of framework includes the definitions of component versus a logical process. Whereas the component exchanges information through input/output ports and not events, and the components must be configured before the system is compiled. [CS01] This last part is a central consideration in how the compositions of basic and complex models work, and is shown in how the auto-generated code and the simulation engine in this project are processed.

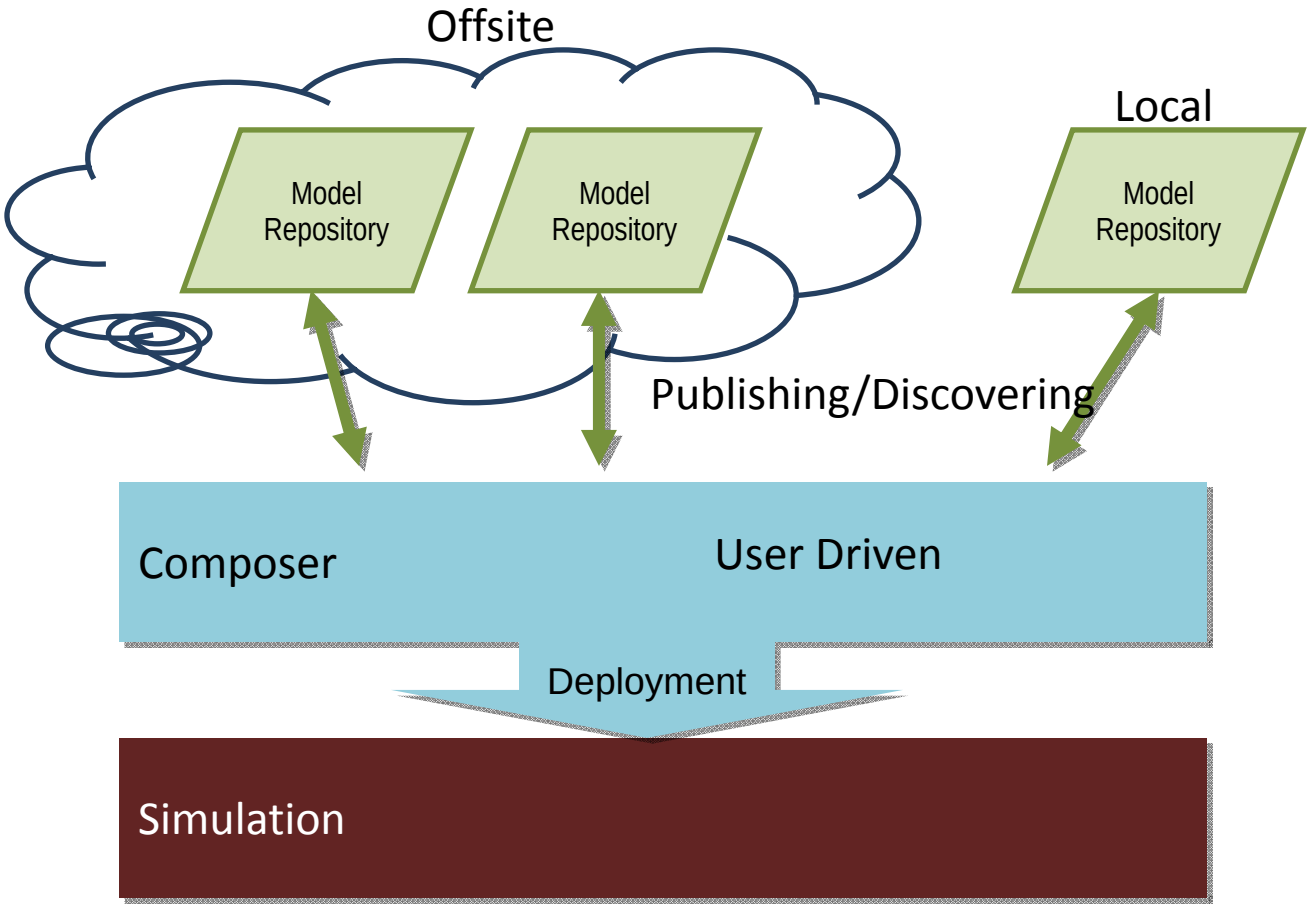

Figure 1 SimAAS Environment

This project is implemented on a single Platform as a Service system with all model contained within the local repository.

Discussion

Cloud computing has been defined to include the following layers. Infrastructure-as-a-Service describes how the basic resources are provided such as data storage, data management, and network connectivity. Platform as a Service describes environments where virtual machines are setup and maintained to provide toolkits for development, distribution, and financial services. Finally, there is the Software-as-a-Service that describes the software available to be used by tenants and tenant's users that is maintained by other organizations and can be rented/bought to accomplish the tenant's needs.[RW11] One concern pointed out by Rajaei and Wappelhorst is that interoperability of cloud services is not supported.[RW11] This project takes the Software-as-a-Service and transforms this to a framework that can be utilized for distributed simulations where a

tenant can create complex simulations based on models built by other tenants. In addition once these models are created then compositions can be created to run in a distributed simulation environment where specific services can be selected to allow the simulations to communicate with other simulations through protocols such as Distributed Interactive Simulation or XML. The concept for the Simulation-as-a-Service is to allow for interfaces to also be selected to allow for specific user visualizations to be used to replicate the view that the tenant wants to give the simulation user such as an interactive 2D versus a 3D view of the simulation run. Primary factors involved in cloud development deal with availability and security which are consistent with any use of a network based system.

According to "Towards A COTS-Based Service-Oriented Simulation Architecture" paper there are three major areas of research into how distributed modeling and simulation systems will work. One is the architecture driven approach as defined by systems like the High Level Architecture, the second is middleware driven such as having a intermediate translator that pulls in disparate protocols and translates them to the other systems, the third is the component based driven layouts where the tendency is to have expert builders who have to be experts in the designing and building of many different systems. This paper proposes the Service Oriented Simulation Architecture (SOSA) where the objective is to build more loosely coupled pieces than represented in most simulation architectures. [GY07] Another Service Oriented Architecture simulation approach is described in the paper "Integrating HLA and Service-Oriented Architecture in a Simulation Framework". This paper explains the FCINT simulation engine and integrates HLA with the FCINT Simulation Framework. This framework uses dynamic application composition, runtime behavior and performance, plus shows the concept of dynamic collaboration within the simulation framework.[DS12]

Another framework proposed for using a service oriented architecture approach focused on loosely coupled services that were directed at the end user. This framework involves the use of web-enabled applications, tools, and resources to build a distributed game simulations architecture based on usefulness, usability, and usage. [HJ04]

Publishing and Discovery

One of the main objectives of this project is to investigate the publishing and discovery relationship between models and a simulation in a Simulation-as-a-Service environment. A simulation environment was developed with an event sequenced simulation engine and along with several models. These models were written to allow for a Composer to include them at a user's discretions into an executable simulation. In order to do this the Composer must discover what a model publishes.

The process of developing a working Composer/Simulation system allowed for the enumeration for some of the basic information that needs to be published. For this project the publishing mechanics were manual. The following solutions will need to be studied in more detail. One possible publishing mechanism is for the model to be responsible to publish information to the composer. An alternative publishing mechanism would be that the composer would actively discover information about the model. For the purposes of this project this will not be covered in this paper. There are four basic areas or information grouping needed for the Composer to allow a user to build a simulation. These areas are: model identification, event linkages, linkage to other models, and data requirements.

The model identification is a set of tags and descriptions necessary to indentify a model uniquely. For this system the published information consisted of a model name, the model type, and a description. The model name is the code ready handle of the model. This name can be wrapped in the appropriate syntax to be called by the simulation engine and is used at code auto-generation time. Paired with the model name is a user understandable long name. The long name is used by the Composer to display model selection. The model type classifies the model for both Composer display and auto-generation. Model types are described in the appendixes below. The item of the model identification is the description. The description is a paragraph length dialog explaining to the user what this model dose. The Composer displays this information for the user.

The event linkage is the name of the scheduled event for this model (if any). This name will be used by the composer to create the auto-generated file *ExecuteEvent* and must accurately reflect the model's syntax.

A model must also publish the linkages to other models needed. Currently this is used to include other models in the build and these interactions must be negotiated between model developers.

The last publishing requirement is a description of the data needed to run the mode. The composer uses this description to prompt the user for information in the entity definition phase. Further research can be done with regard to other publishing and discovery mechanisms. One promising area is the publishing and discovery process that is managed by the Universal Description Discover and Integration (UDDI) registry method based on XML. [SZ09]

Composition

Use-Case Diagram

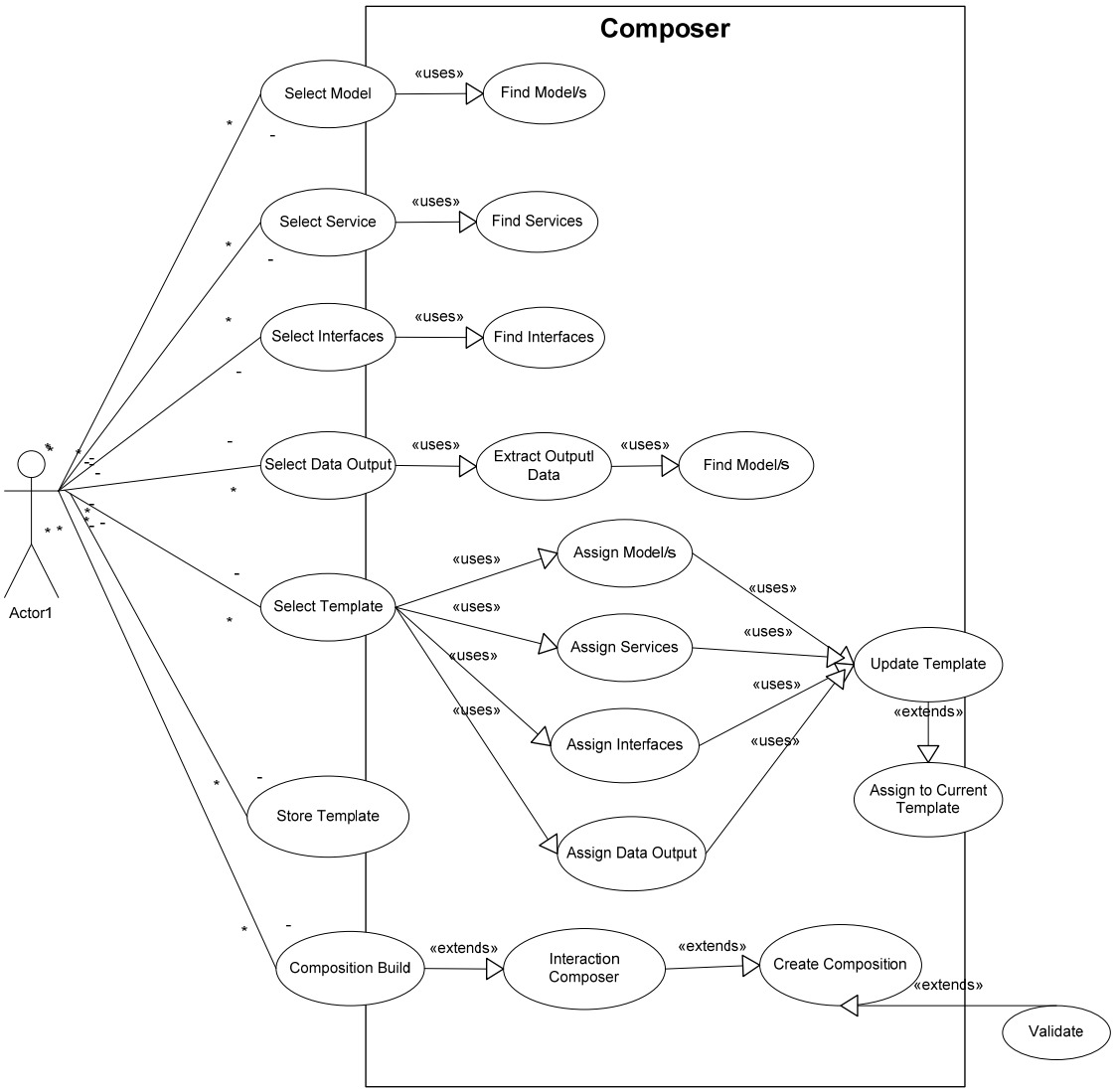

Figure 2 User Interactions with the Composer

Composer

During the initial investigation into how a composer would work in a multi-tenant environment several requirements were discovered.[TS11] One was that the system needed to be able to restrict other tenants from using models that another tenant did not want to share(protection of tenant sensitive information), a second was that the internal working of a model needed to be isolated from the tenant in order to reduce the complexity,

another item was that the system should follow the software-as-a-service paradigm where there exists only one version of the code base that is accessed during the creation of auto-generated code and is based on the object model agreements in place. This will help with the complexity of code maintenance involved with the various models and the simulations engine. A tenant is always guaranteed to be running the latest models and simulations engine during the creation of the composition. See appendix A and appendix B to look at the simulation engine details and simulations engine model agreement.

During the literature search a paper by "SimSaaS:Simulation-as-a-Service" proposed an overall framework. This framework covered the multi-tenancy configuration model, along with how the multi-tenancy works during the simulation run-time.[TS11] This project focuses on how the composer would interact with a tenant and how the composer interacts with the auto-code generation and the running of the resultant simulation code. The composer consists of multiple parts. These parts include the following: A composed template describing all services, interactions, complex models, data output; a process to build, edit, delete, and share complex models created from models that have been published and discovered from the same tenant or other tenants within the SimSaaS framework; A process to create, edit, and share this composition; A process to create, edit, and delete simulation data through the creation of entities based on the complex model types created by the tenant; A process to auto generate code based of the complex models created by the tenant to integrate the complex model behavior into the simulation engine; and a process to run the simulation and save the simulation results based on the tenant.

A point to concentrate on with regard to the composer is that it is a tenant based system as inherent in most Software-as-a-Service systems. This allows flexibility in creating solutions that are based on what the user needs. In addition there should be the ability to collaborate from multiple locations on the same composition process so that individuals will be able to give real-time input into the composition process. The composer depends a great deal on the service level agreements/federation agreements to know what information to supply to the tenant doing the composition. Through the service level agreements inputs/outputs, and data requirements are defined and agreed upon by the tenants/users of the simulation-as-a-service framework. The composer discovers other created compositions, basic and complex models, services, interfaces, and data output methods that have been created by this tenant and also these items that have been shared by other tenants. This is a data driven approach where during the publishing phase and according to the service level and object model agreements a tenant will publish the required information concerning their respective services, interfaces, and models. A data repository contains this information and is accessed by the composer to provide a given tenant available options to build a composition from. A primary distinction from most simulations in use today is that the SimSAAS system allows individual models to be built

and maintained by other tenants without the user being required to store and maintain a given model. Maintaining a model requires the owner to work within the service level agreement to provide updates, fixes, and new code. This is analogous with current "cloud" systems that are coming into play today, such as the ability to store and view photographs. The viewing software is generally referred to as a thin client similar to what was used in the early computing days in what was called a remote terminal, and the data is stored on externally controlled and maintained hardware server farms. The following diagram shows how the Composer would fit into an overall Simulation-as-a-Service view.

There is parallel work going on with Simulation-as-a-Service. Specifically systems that describe cloud based simulation. Most of this work is focused on scheduling, security, infrastructure as a service (IAAS), and platform as a service (PAAS). For instance scheduling methods for Cloud based simulation are described in the article "Cloud-based Simulation: the State-of-the-art Computer Simulation Paradigm" [LH12]. Another similar

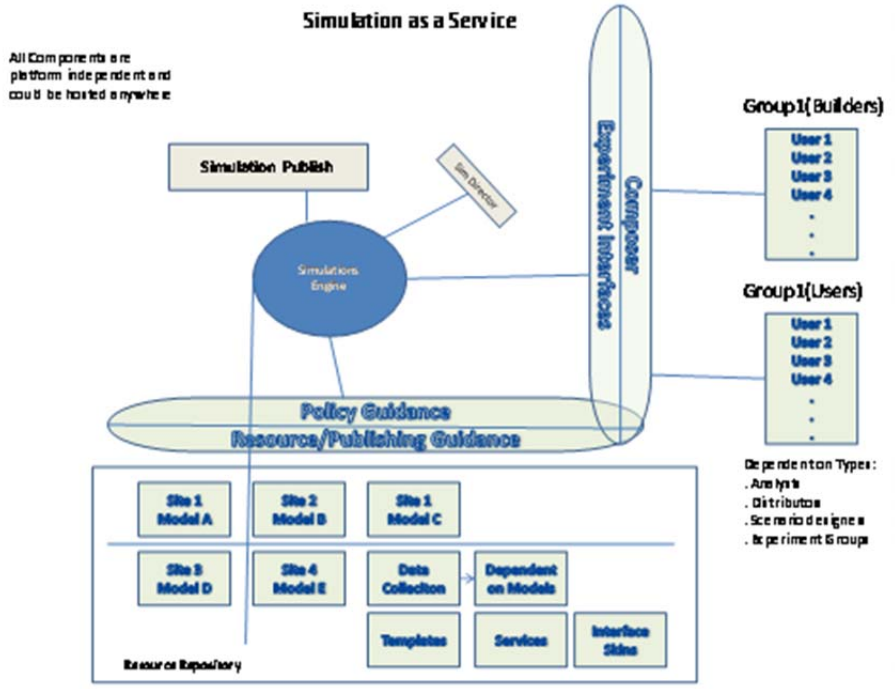

Figure 3 Simulation-as-a-Service View

Simulation-as-a-Service modeling framework is described in “A Simulation Framework for Service-Oriented Computing Systems” is called the Discrete event System specification (DEVS) which is oriented to the Service Oriented Architecture methodology.[SY08]For this project demonstration architecture it was determined to locate a web hosting system that also hosts database resources as well as tools to interact with these resources. This in effect will provide the IAAS and PAAS portion of our demonstration. Multi-tenancy will be handled through a simple session control mechanism that uses MD5 generated strings to differentiate tenants and track models, interfaces, services, and compositions that can be shared or not shared by other tenants. Further research will need to be conducted to determine the collaboration mechanisms that would be the most beneficial to the scalability and real-time coordination that large scale simulations require. In addition the services and interface pieces are not trivial and will require extensive research and development to provide the needed services such as different transport protocols such as Distributed Interactive Simulation (DIS) or High Level Architecture – Federated Object Model (HLA-FOM) variants. The services that are a precondition for the composition of the complex models in this project are related to the coordinate/terrain service, timing service, and data services that have been built to allow for the demonstration of the composer concept. Additional complex tasks will be the interface design and usage for human in the loop Simulation-as-a-Service participants. These services will all separate users/tenants to do group collaboration across this framework. For the purposes of this project the following tools were used to construct the composer demonstration. Database storage MySQL 5.5.28, Database administration: phpMyAdmin 3.4.11.1, Dynamic webpage construction : PHP 5.2, web client side validation and processing: Javascript:1.8.5, and Jscript version 9.0

Deployment

The deployment level of this system is a functioning simulation. The simulation consists of a simulation engine (SE) and a composeable group of models and services.

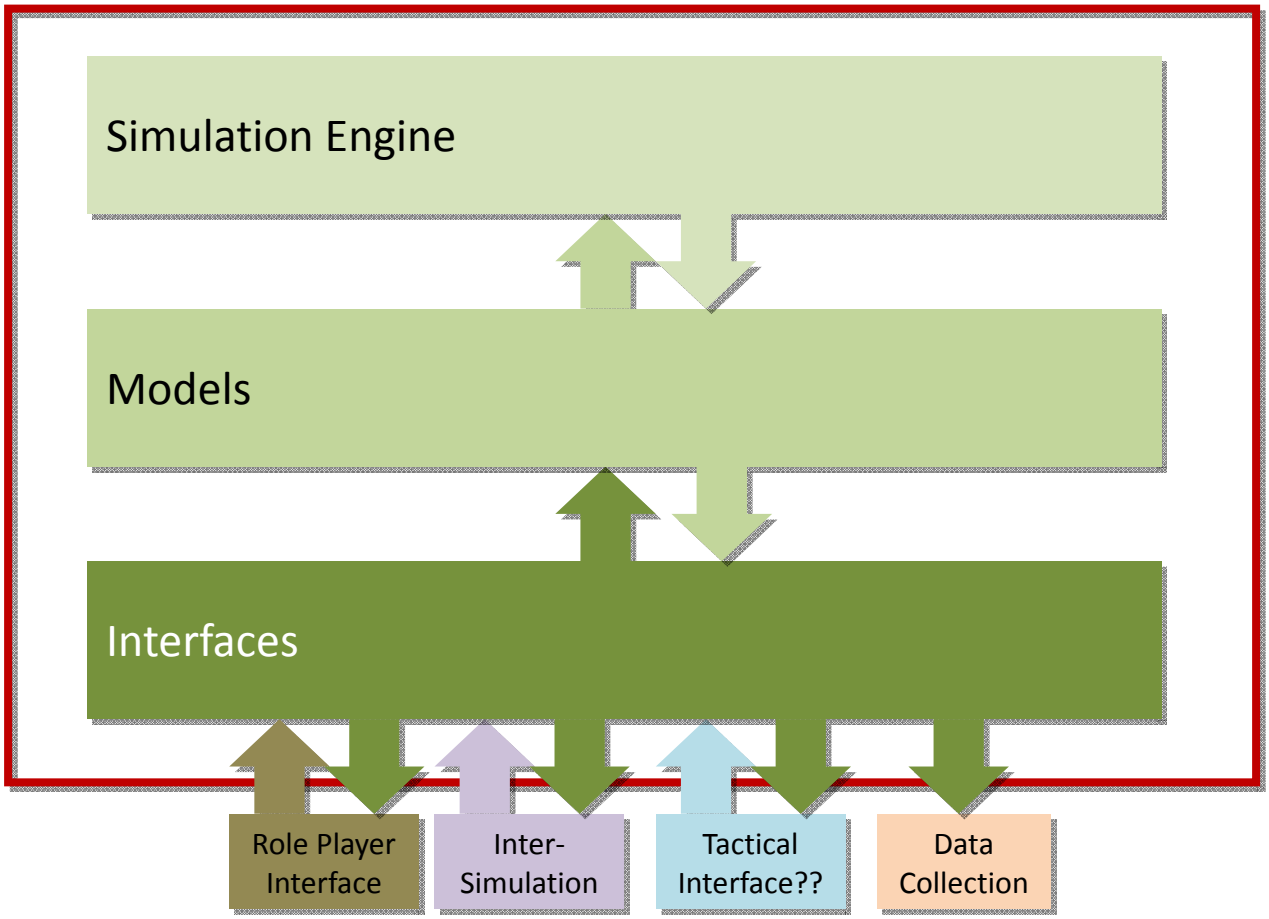

Figure 4 Simulation Environment

Models

For this project 14 models of 5 types were developed. The table below is the list of the available modes for the current Composer to Simulation Engine lash up. Entities are composed of several models with the minimal being a physical and a movement model.

Model	Type	Description
Null Model		
Behavior Aggressor Air	Behavior	This model will order the entity to follow a given path in the air
Behavior Aggressor Ground	Behavior	This model will order the entity to follow a given path on the ground
Behavior Observer	Behavior	This model will order the entity to search a given area
Data Collection Observer	Data	This model will collect data on observer

		acquisitions
Movement Air	Movement	This entity will follow a given path in the air
Movement Ground	Movement	This entity will move to a given point on the ground
Movement Null	Movement	This entity will not have a location within the game
Movement Stationary	Movement	This entity will not move. It will remain stationary at its initial location
Physical Life Form	Physical	This entity is a Life Form
Physical Null	Physical	This entity will not have a Physical appearance. For example the entity may be off the map or a decision node
Physical Vehicle	Physical	This entity is a vehicle
Sensor Air	Sensor	This entity will search to a given area in the air
Sensor Ground	Sensor	This entity search a given area on the ground

Physical modes represent the material aspect of an entity and are one of the two required models. This model type describes the shape of the entity and its side (red or blue). A null model is provided for entities that do not appear on the map.

Movement models are responsible for moving an entity from one point to another. These models are commanded where and when to move by behavior models.

Behavior models control movement model and sensor models. Command them when and where to move to or when and where to search. Behavior models will also link with data collection services.

Sensor model search the geo-location table and report out to its controlling behavior mode the results.

Data collection models (or services) are responsible for reporting results from behavior modes (currently only search results).

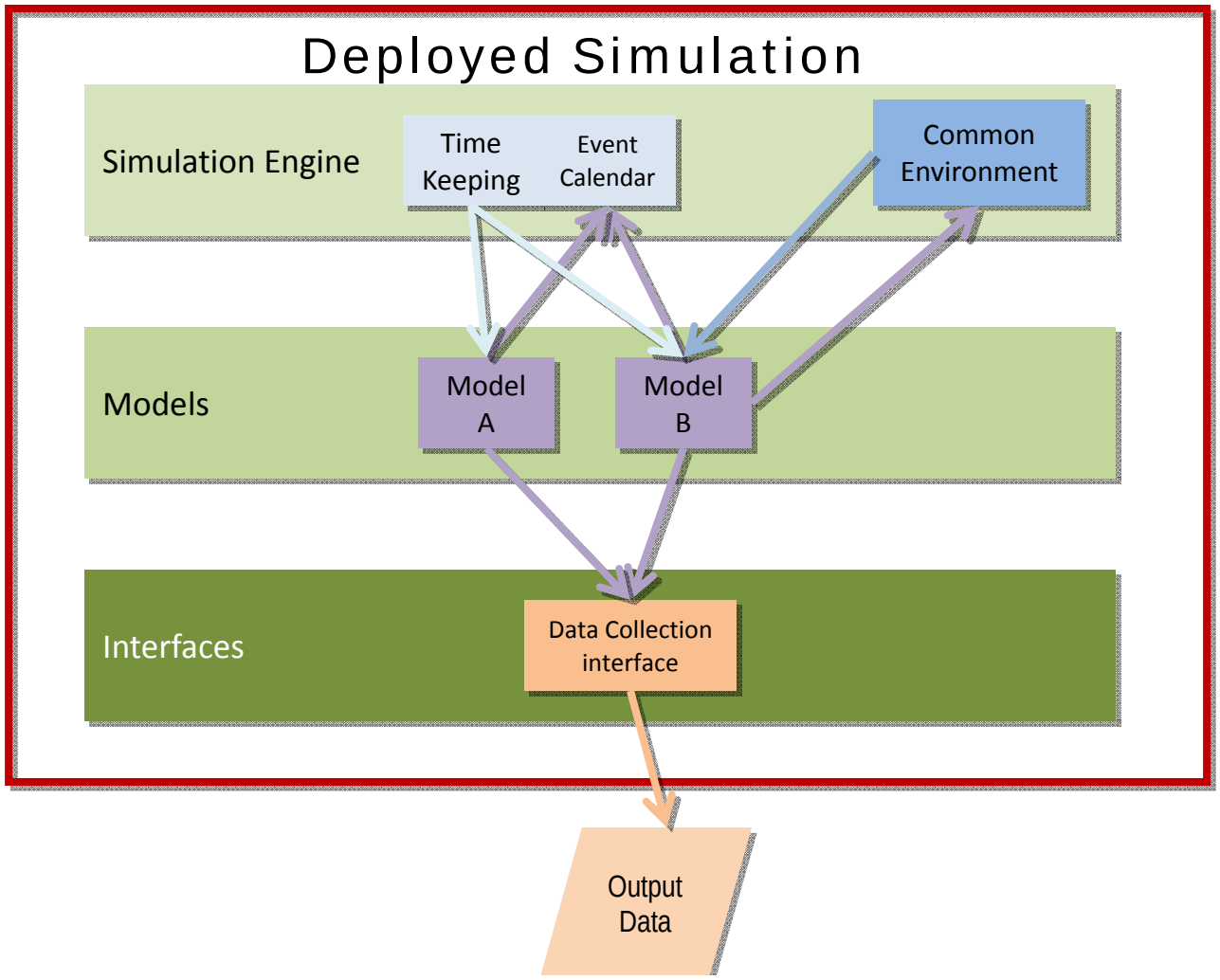

Figure 5 Inter-Model Interactions

Conclusion

In conclusion this project has focused on the composition of basic and complex models into a system where a tenant controls what models are present in simulation built to the tenants needs. This project has demonstrated the ability and the actions required in a multi tenant environment for this process. The investigation has shown that there needs to be further research into the areas of data management used to store the information on model composition. There are other component based systems that have done research into alternate methods such as SOSA, FCINT simulation, or web based services to describe the Simulation-as-a-Service paradigm that are worth further study. In addition more work needs to be done in IAAS, and PAAS domains to work requirements needed for large scale distributed simulation. Lastly, security will become increasingly important if the architecture is used among tenants that do not want to share information

with other tenants which leads to more research being done into encryption needed for all aspects of the Simulation-as-a-Service architecture and framework.

Demonstration Environment

A demonstration of the current state of the Composer is available through the web at http://www.composer.simaas.us/SimSAAS_registration.php.

List of References

[CB12] Tony Clark and Balbir Barn. "A Common Basis for Modeling Service-Oriented and Event-Driven Architecture". Proceedings of ISEC '12. Feb 22-25 2012:23-32.

[SY08] Hessam Sarjoughian et al."A Simulation Framework for Service-Oriented Computing Systems." Proceedings WSC 2008. Dec 2008:845-853.

[BO00] Perakath Benjamin et al. "A Model-Based Approach for Component Simulation Development". Proceedings WSC 2000, Dec 2000:1831-1839.

[GG10] Guo Gang et al. "Architecture and Standard Proposals for Next Generation Modeling and Simulation". Proceedings GCMS '10, July 2010:12-19.

[HP95] John Hamilton and Udo Pooch. "An Open Simulation Architecture for FORCE XXI." Proceedings WSC 1995, Dec 1995:1296-1303.

[BB09] E Baydogan, S Mazumdar, and L Belfore II. "Simulation Architecture for Virtual Operating Room Training". Proceedings SpringSim '09, Article No. 25, 5 pages.

[TS11] Wei-Tek, et al. "SimSaaS:Simulation Software-as-a-Service." Proceedings of the 44th Annual Simulation Symposium, ANSS '11. 2011:77-86.

[HJ04] S Houten and P Jacobs. "An Architecture for Distributed Simulation Games", Proceedings WSC 2004, Dec 2010:2081-2086.

[CS01] G Chen and B Szymanski," Component-Oriented Simulation Architecture: Toward Interoperability and Interchangeability", Proceedings WSC 2001, Dec 2001:495-501.

[GY07] T Gu, N Lo, and W Yang. " Towards a COTS-Based Service-Oriented Simulation Architecture", Proceedings of the 2007 summer Computer Simulation conference SCSC '07, 2007:1128-1135.

[RW11] H Rajaei and J Wappelhorst. "Clouds & Grids: A Network and Simulation Perspective", Proceedings of the 14th Communications and Networking Symposium CNS '11, 2011:143-150.

[DS12] M Dragoicea, L Bucur, W Tsai, and H Sarjoughian. "Integrating HLA and Service-Oriented Architecture in a Simulation Framework", Cluster, Cloud and Grid Computing 2012, CCGRID '12, 13-16 May 2012:861-866.

[TS09] W. T.Tsai, H Sarjoughian, W Li, and X Sun. "Timing Specification and Analysis for Service-Oriented Simulation." Proceedings of the 2009 Spring Simulation Multi-conference, 2009:Article 51

[RU07] Mathias Rohl, Florian Marquardt, and Adelinde Uhrmacher. "Exploiting Web Service Techniques for Composing Simulation Models." Proceedings of Winter Simulation Conference 2007, WSC'07, Dec 2007:833-841

[EG06] Erek Gokturk. "Towards Simulator Interoperability and Model Interreplaceability in Network Simulation and Emulation through AMINES-HLA" Proceedings of Winter Simulation Conference 2006, WSC'06, Dec 2006:2170-2179

[WL08] Wenquang Wang, et al. "Service-Oriented High Level Architecture" European Simulation Interoperability Worksop 2008

[PC10] Pau Fonseca I Casas. "Using Specification and Description Language to Define and Implement Discrete Simulation Models" Summer Simulation Multi-conference 2010, SumerSim '10, 2010:419-426.

[SZ09-1] Chungman Seo and Bernard Zeigler. "Interoperability between DEVS Simulators using Service Oriented Architecture and DEVS Namespace" Proceedings of the 2009 Spring Simulation Multi-conference, SpringSim '09, 2009: Article No. 157.

[SZ09-2] Chungman Seo and Bernard Zeigler. “Automating the DEVS Modeling and Simulation Interface to Web Services.” Proceedings of the 2009 Spring Simulation Multiconference, SpringSim ’09, 2009: Article No. 158.

[LH12] Xiaochang Liu, et al.”Cloud-based Simulation: the State-of-the-art Computer Simulation Paradigm”. Principles of Advanced and Distributed Simulation Workshop 2012, July 2012:71-74.

Appendix A - Simulation-As-A-Service Simulation Engine

The Simulation Engine (SE) is an event sequenced stochastic system. It is designed to allow models developed external to the simulation engine to be incorporate. These models must adhere to the SE Federated Model Agreement. Additionally each model must publish the required information for the Composer to discover in order to allow a user to select the model.

The Simulation Engine consists of 4 areas or sections: initialization, scheduling and time keeping, services, and a common environment.

Models interface with the SE through auto generated code. Each of the above areas contains code generated at composition time. Models types will be described in detail in the SE Federated Model Agreement documentation (Appendix B).

The SimAAS Simulation Engine is implemented in gnu c++. It is organized into four compartments (SE, Services, AutoGen, and Models) each with its own file structure.

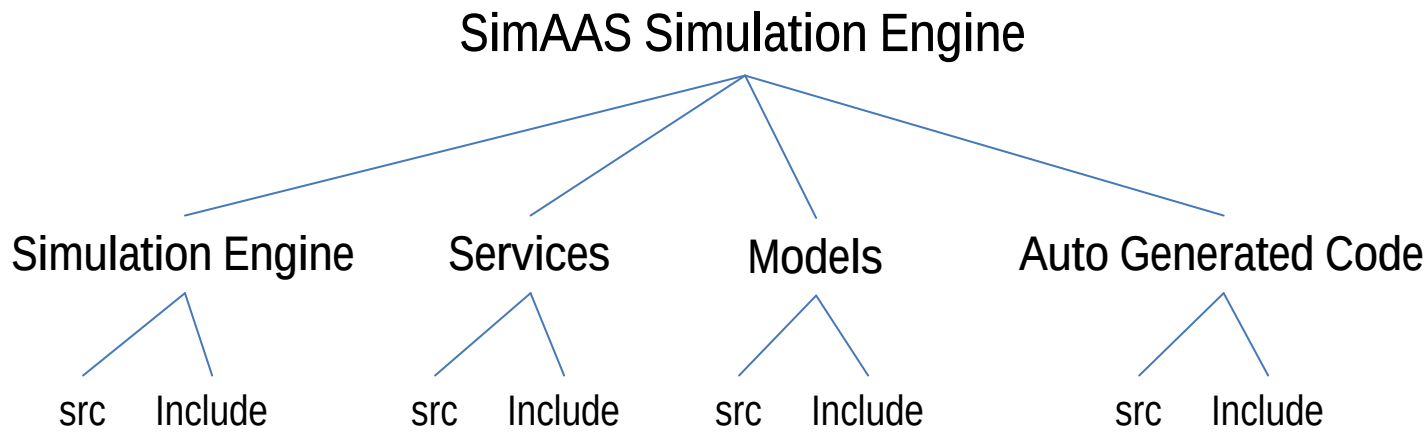

Figure 6 SE File Structure

Use Case

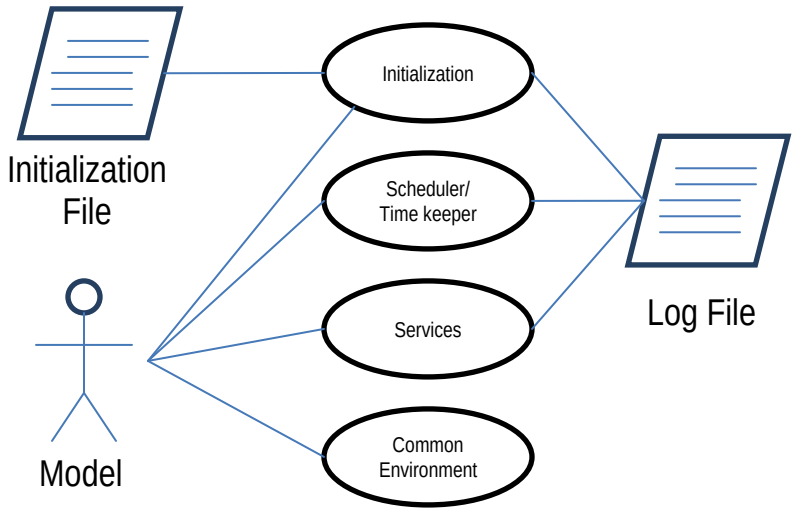

Figure 7 SE Use Case

Note: the terms *model* and *entity* are used interchangeably below. *Model* represents the class and associated member functions while *entity* is an instantiation of a model or group of models containing the current state.

Initialization

The Simulation Engine (SE) main will process run time input arguments, instantiate the Simulation Engine class and kick the Time Keeper off to run the simulation. The initialization

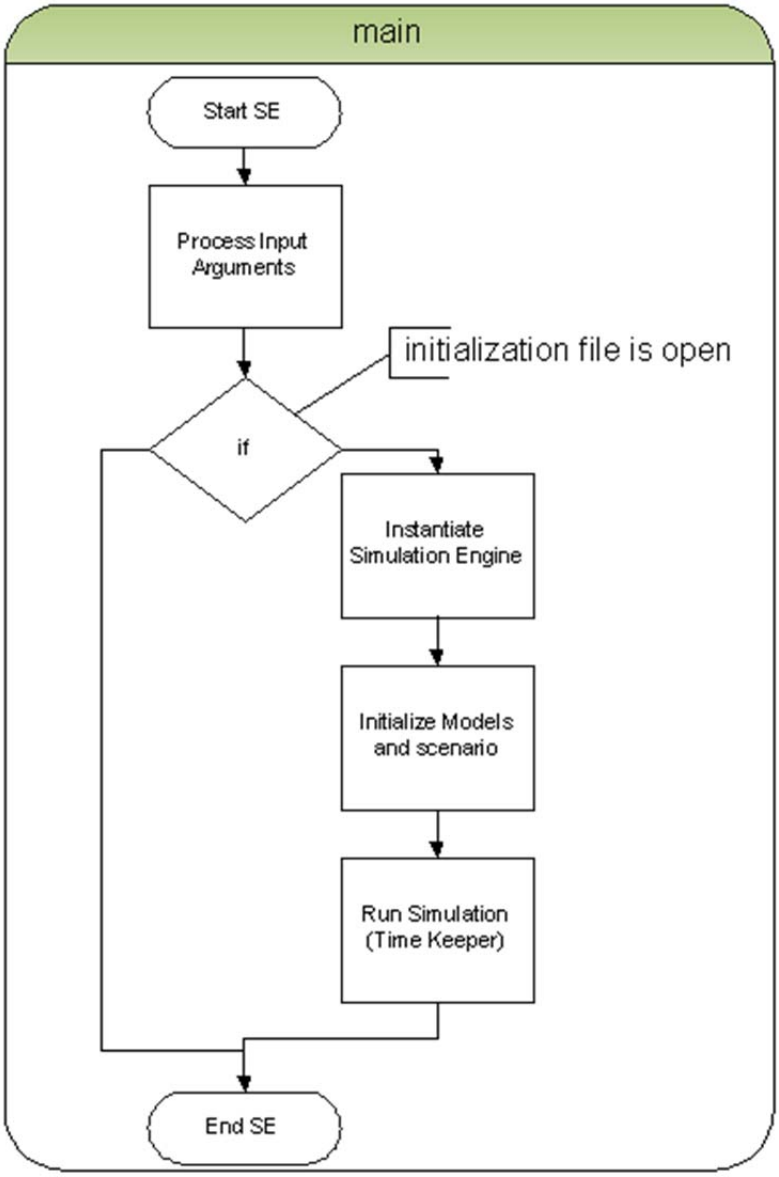

Figure 8 Workflow for main

file is open for latter processing in the initialize models section (see appendix C for an explanation of this file).

Runtime Input Arguments

Currently the only run time input argument is '-I' for specifying the initialization file. It is of the form ***–I file_name***. If the '–I' option is not present then a default file of *test.ini* will be used (this

is useful for debugging). The simulation can be run manually from the main directory on a Linux platform by:

bin/SE_linux –I test.ini > run.log

A user will run the simulation from a user interface and will not be aware of this command.

Instantiate Simulation Engine

The Simulation Engine is a Singleton class and must be invoked through the Instantiate function. The constructor is private and cannot be called externally from the class thus ensuring only one instants of the SE will be available. The global pointer SimEngine gives each model access to any function needed. Since entities are instantiated by the Simulation Engine this variable will available to all models invoked by the initialization process using information from the initialization file.

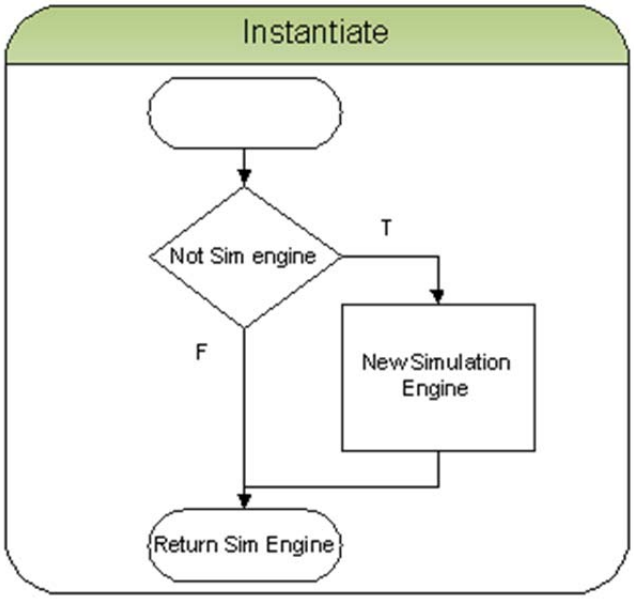

Figure 9 Instantiate Workflow

Initialize Models and Scenario

The initialization file contains the information necessary to insatiate each entity and its model. The initialization file is generated by the Composer in comma separated formant. The first field is a keyword indicating the format of the record. This file is the main interface between the composer and the simulation. The format of this file is defined in Appendix C.

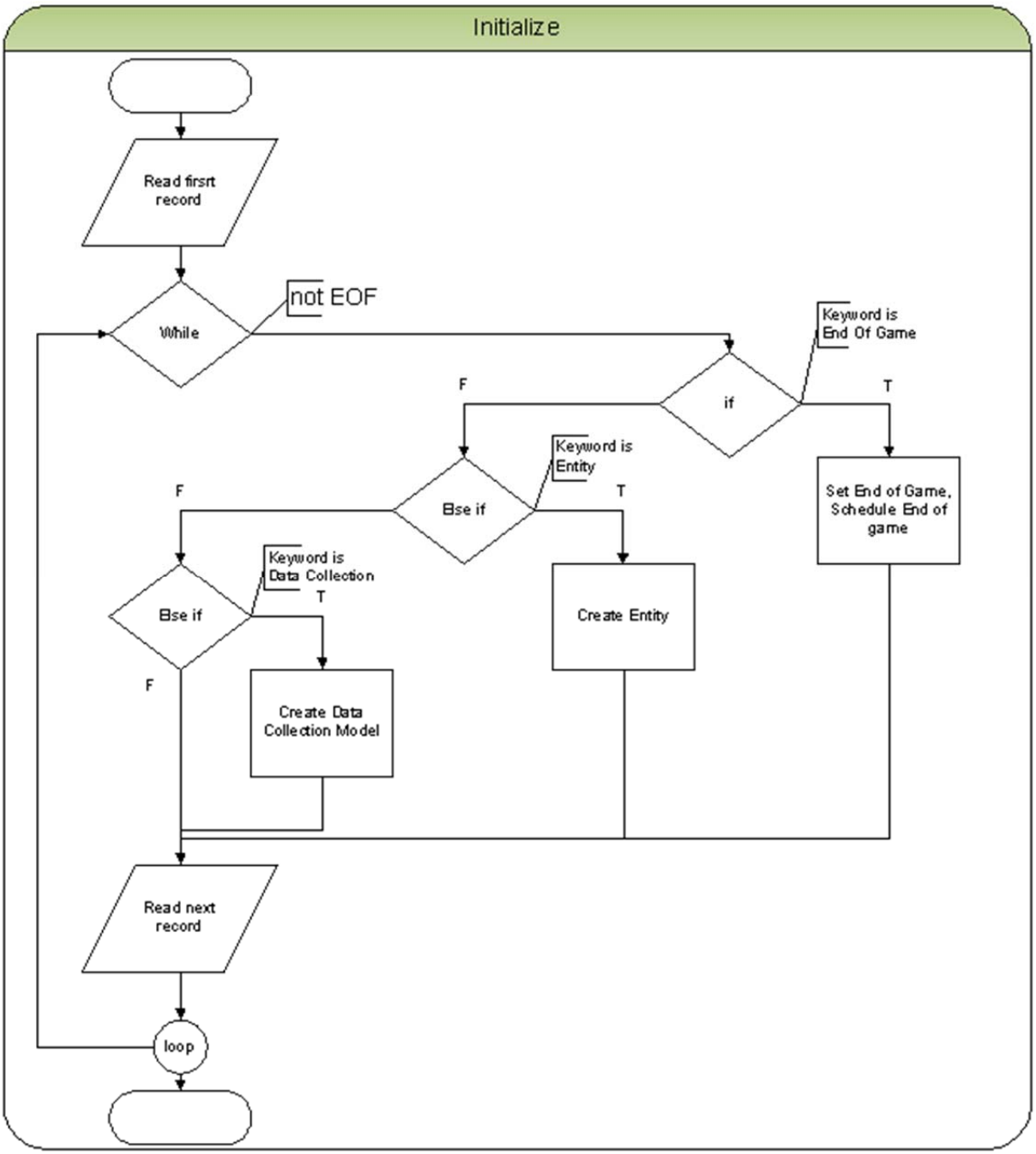

Figure 10 Initialize Workflow

Create Entity

Create Entity uses the composer generated code to call the model constructor for each entity to be instantiated. An entity as a minimum must have a physical and a movement model.

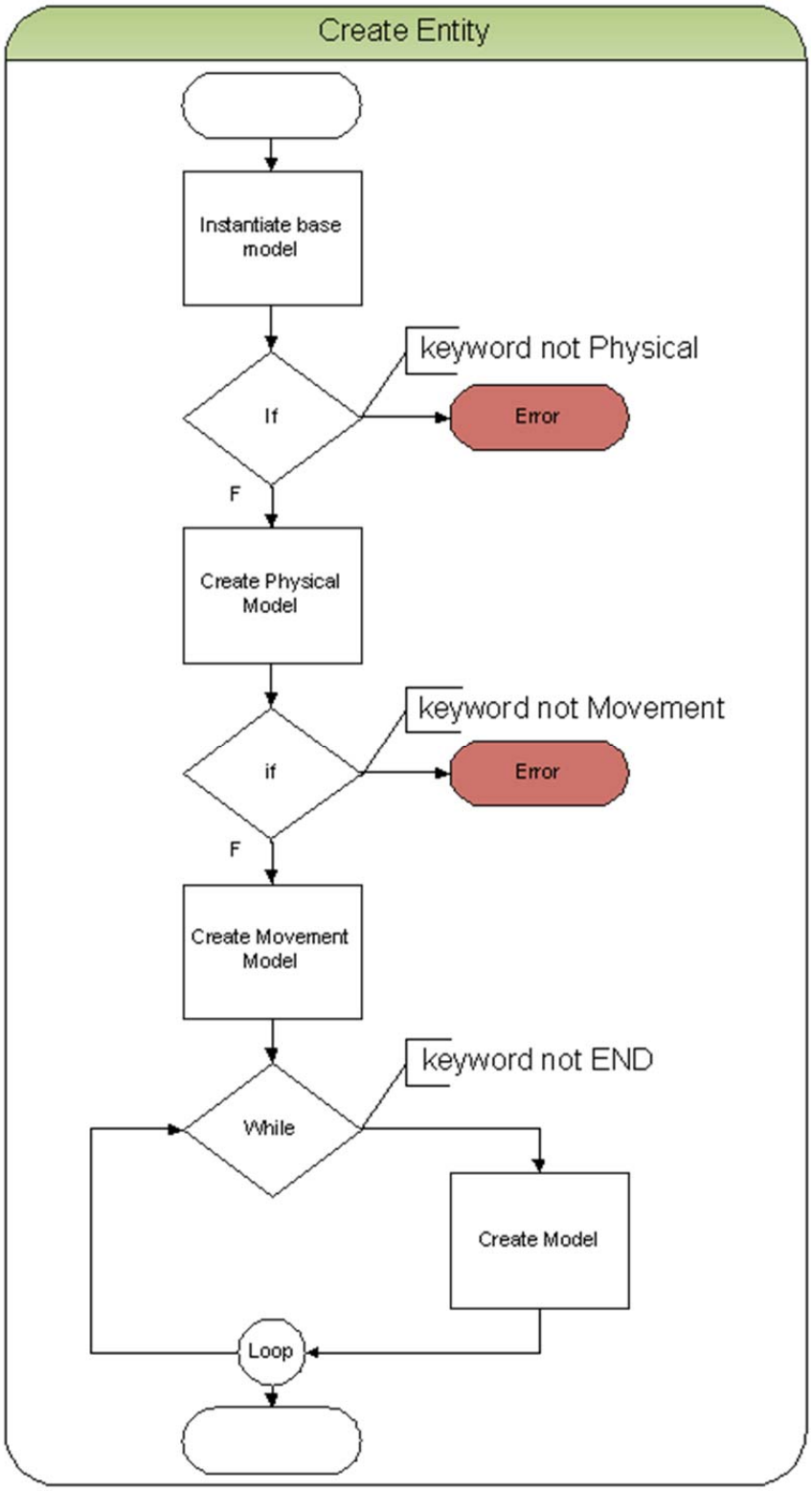

Figure 11 Create Entity Workflow

Create Physical Model

Create Physical Model instantiates one of the two required models that compose an entity. This is an auto generated file created by the composer at the instruction of the user. Physical models represent the real world state of the entity such as its size and shape. The data required by each model is specific to that model and a description must be provided (published) to the composer. Interactions between the Simulation Engine and physical model are described in SE Federated Model Agreement documentation (Appendix B).

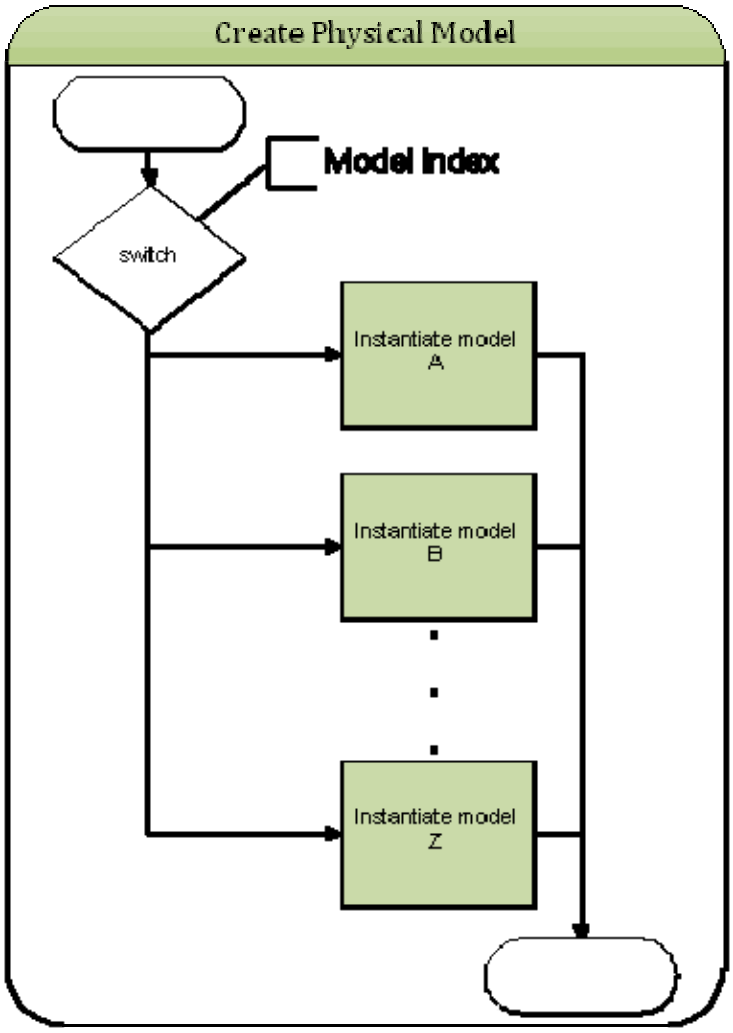

Figure 12Create Physical Model Workflow

The shaded boxes in the diagram above are externally functions to the Simulation Engine. The composer must discover these models and provide the information to the user at composition time.

Create Movement Model

Create Movement Model instantiates the other of the two required models that compose an entity. This is also an auto generated file. Movement models are responsible for maintaining the entity's location in geo-space and how it moves. The data required by each model is specific to that model and a description must be provided (published) to the composer. Interactions between the Simulation Engine and physical model are described in SE Federated Model Agreement documentation (Appendix B).

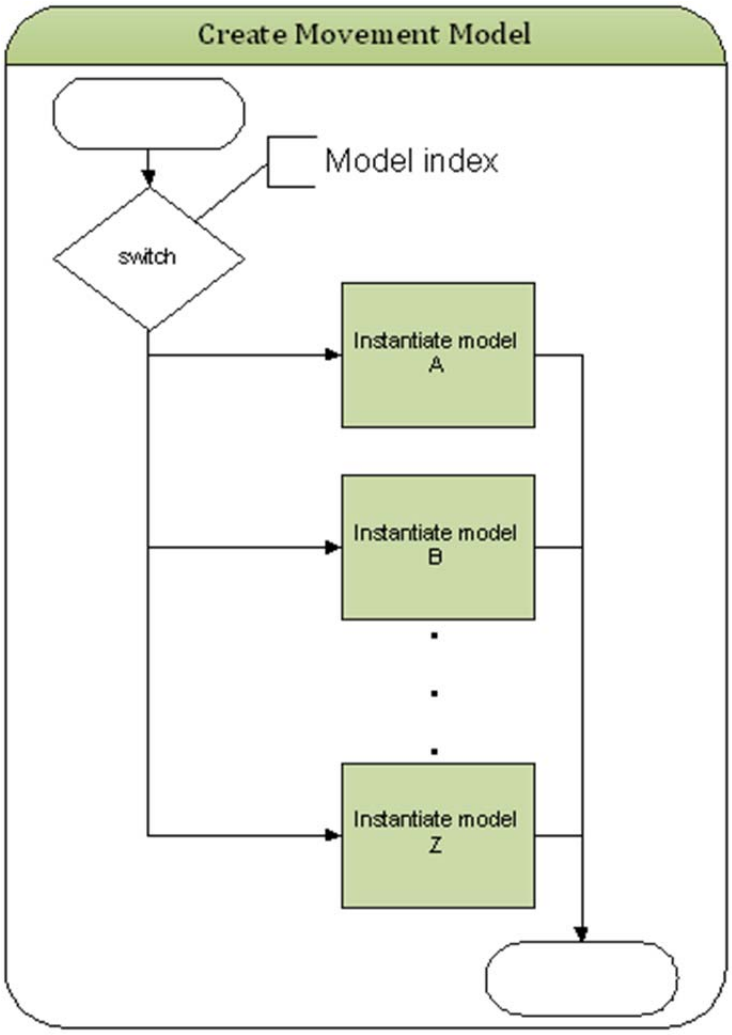

Figure 13 Create Movement Model Workflow

The shaded boxes in the diagram above are external functions to the Simulation Engine. The composer must discover these models and provide the information to the user at composition time.

Create Model

Create Model instantiates the other non physical or movement model that compose an entity. This is an auto generated file. Currently there are three types of models available for selection by the user: Behavior, Sensor, and Data Collection. This system is flexible enough to allow future expansion of model types. The data required by each model is specific to that model and a description must be provided (published) to the composer. Interactions between the Simulation Engine and physical model are described in SE Federated Model Agreement documentation (Appendix B).

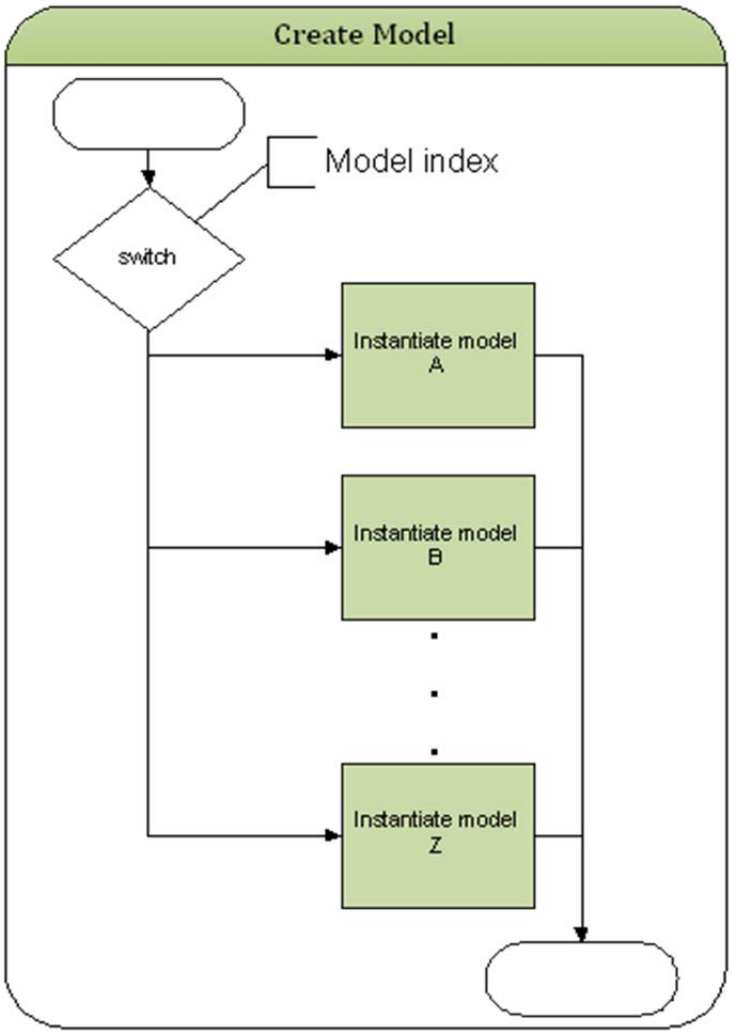

Figure 14 Create Model Workflow

Behavior models control the timing of the ‘when’ and ‘why’ of an entity’s actions. For example a sensor behavior model would command the sensor model when and where to look in the simulation geo-space. Sensor models will quire the geo-location data base. Data logging services will record specific information based on stimulus form other models.

As been stated above the shaded boxes in the diagram above are externally functions to the Simulation Engine.

Create Data Collection Model

Create Data Collection Model instantiates the data collection services. This is an auto generated file.

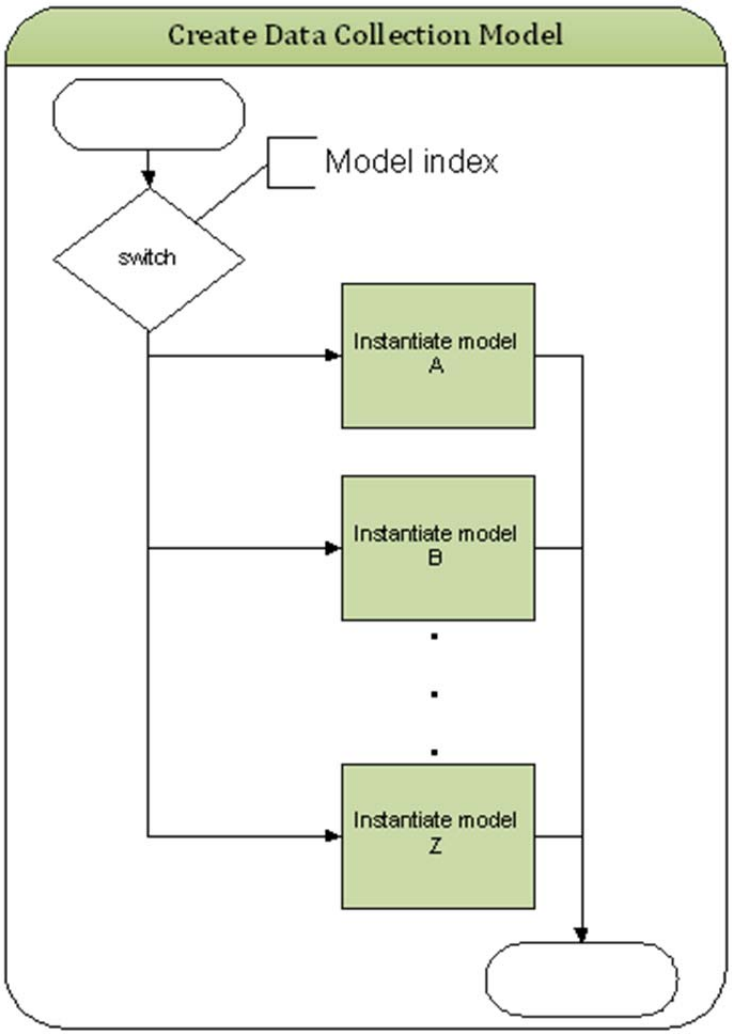

Figure 15 Create Data Collection Model Workflow

Scheduler/Time keeper

The Scheduler and Time Keeper add timing to the Models thus generating a Simulation (Models + time = simulation). The Scheduler /Time Keeper is responsible for:

1. Maintaining the event calendar
2. Maintaining time
3. Calling a model at the proper time

The Time Keeper once called will loop until there are no events remaining in the event calenderer or the end of game event is encountered. The current implementation is a faster-than-real-time event timed sequenced system. The Time Keeper retrieves the next event from the event calendar advances the game clock to the time in the event and then calls the event. Since the current event is on the 'top' of the event calendar no other event will occur before it gaining run-time efficiency. The shaded box in the diagram below (Execute Event) is an auto generated routine.

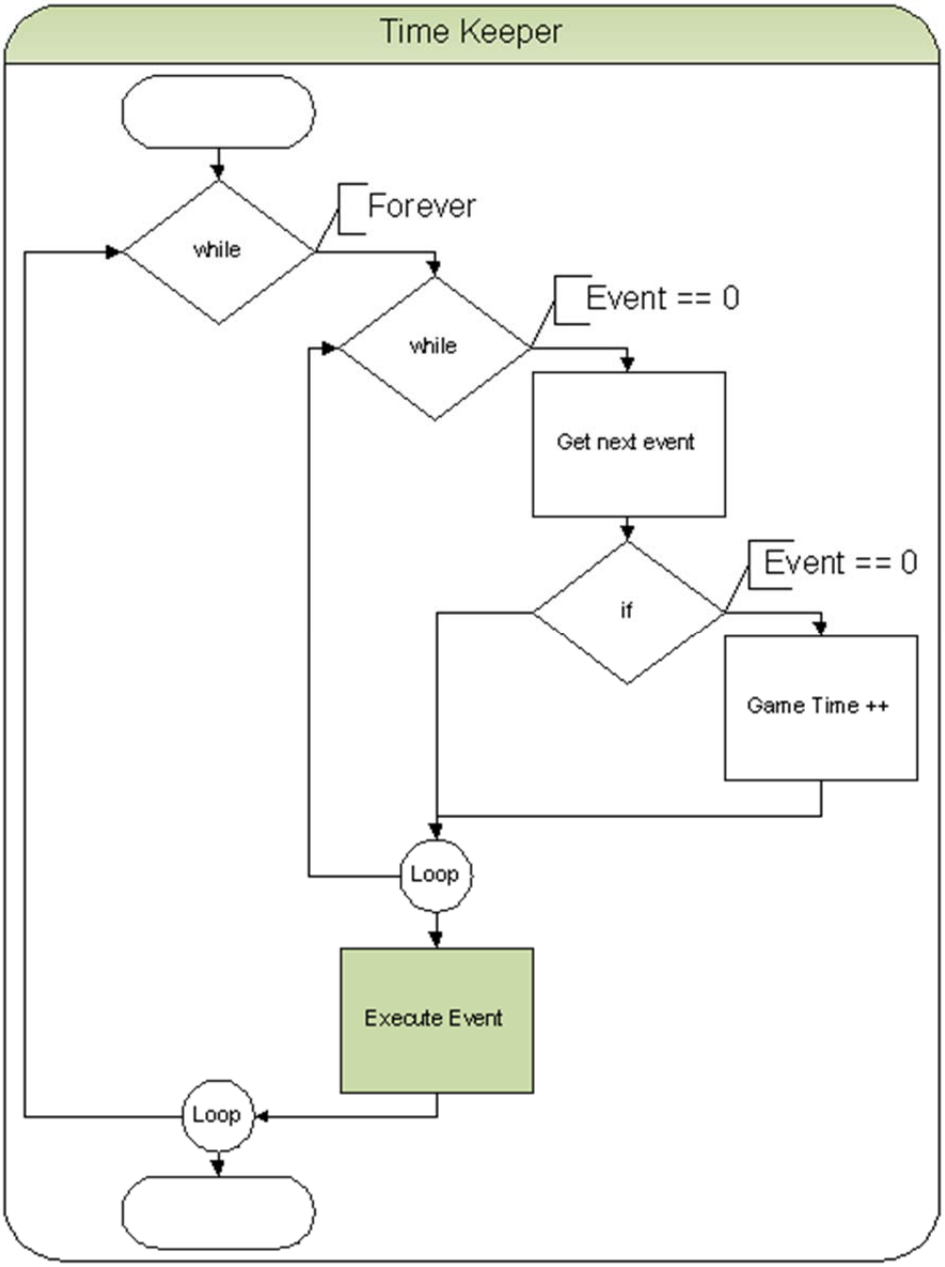

Figure 16 Time Keeper Workflow

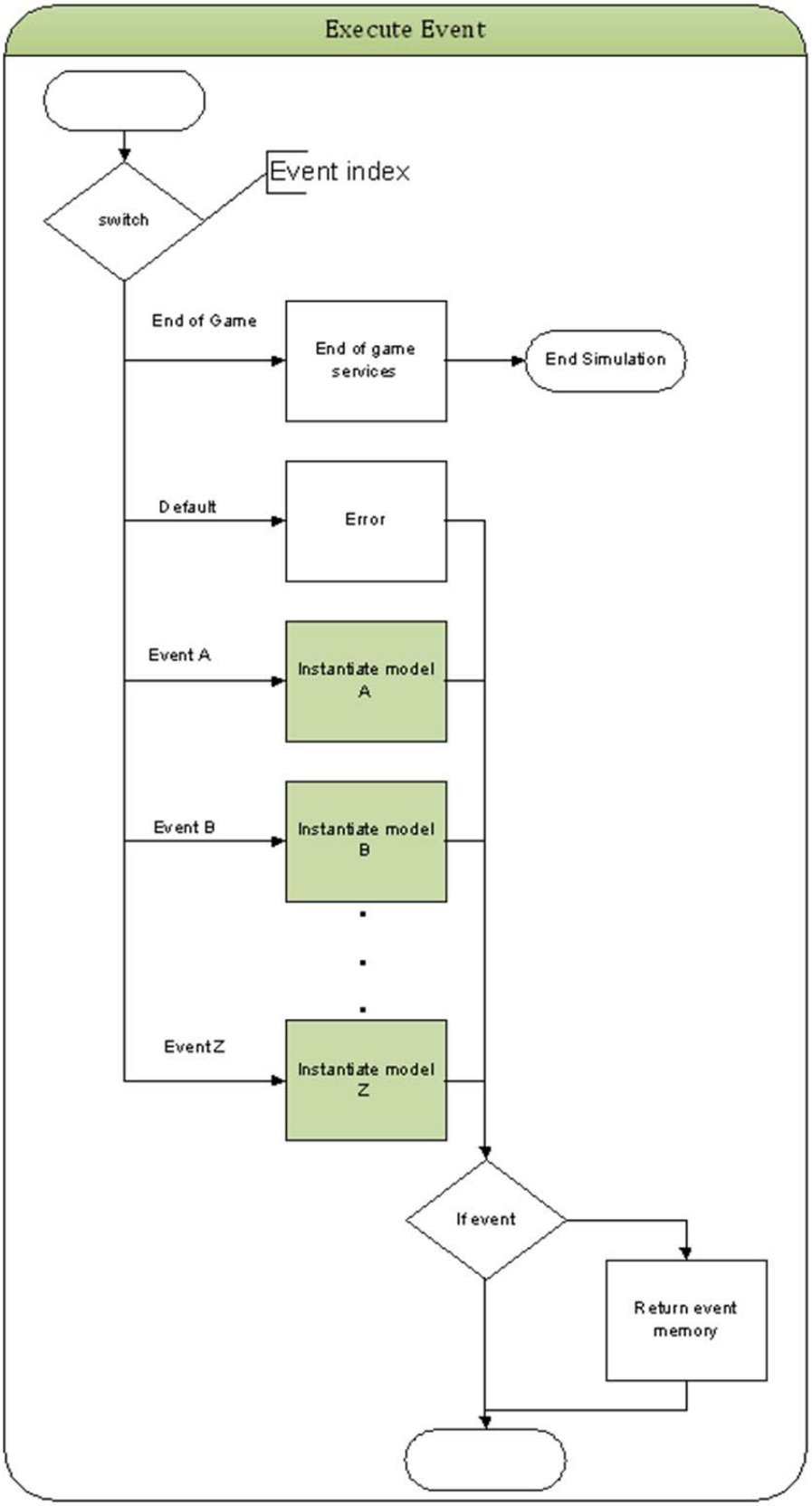

Figure 17 Execute Event Workflow

Scheduling

The SE is a event sequence simulation driver. When a model schedules an event it will do so through the SE Scheduling service. Multiple models can schedule events at the same time, possibly having some user-defined ordering consistent with the priority defined below).

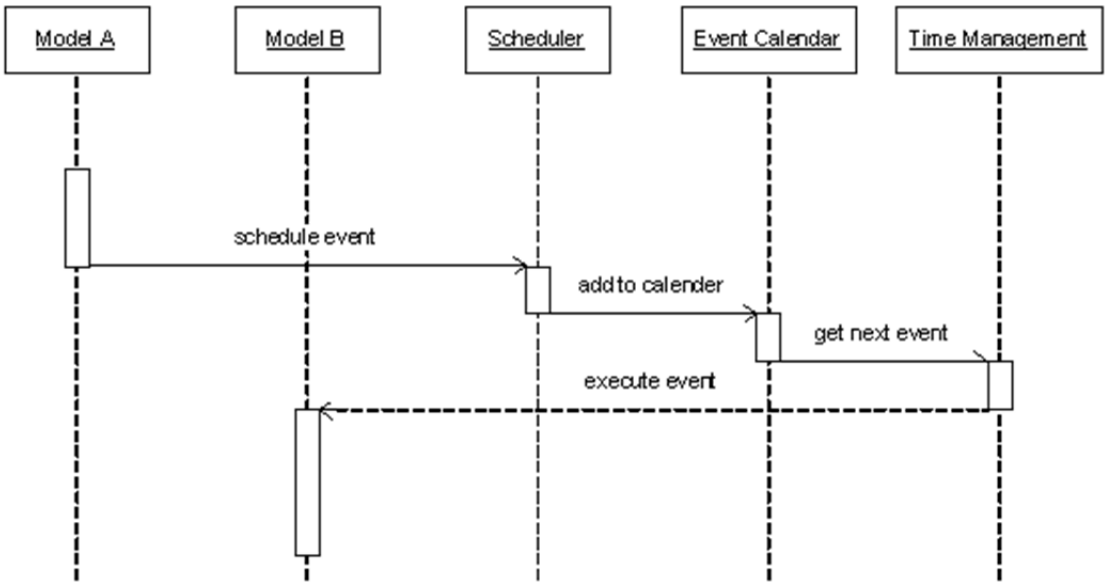

Figure 18 Sequence Diagram for Model Scheduling

Scheduling an event is done by a call to the Simulation Engine member function Schedule. The function:

SimulationEngine:: Schedule(int EventIndex, int Time, int Priority, Void *Event)

Where:

> **EventIndex** is a unique identifier for the scheduled model (in the diagram Model B). The composer must supply this handle to Model A (or any model needing to schedule Model B)
>
> **Time** is when this event will occur (in game time units). This time must be in the future or the Scheduler will drop this event.

Priority is the importance of the event relative to other events occurring at the same time. This is a tie breaker and if two events with the same time and priority are scheduled then the first one scheduled it the first to execute.

Event is a pointer to a block of data required by the scheduled model to function. The structure of the event is unknown to the scheduler.

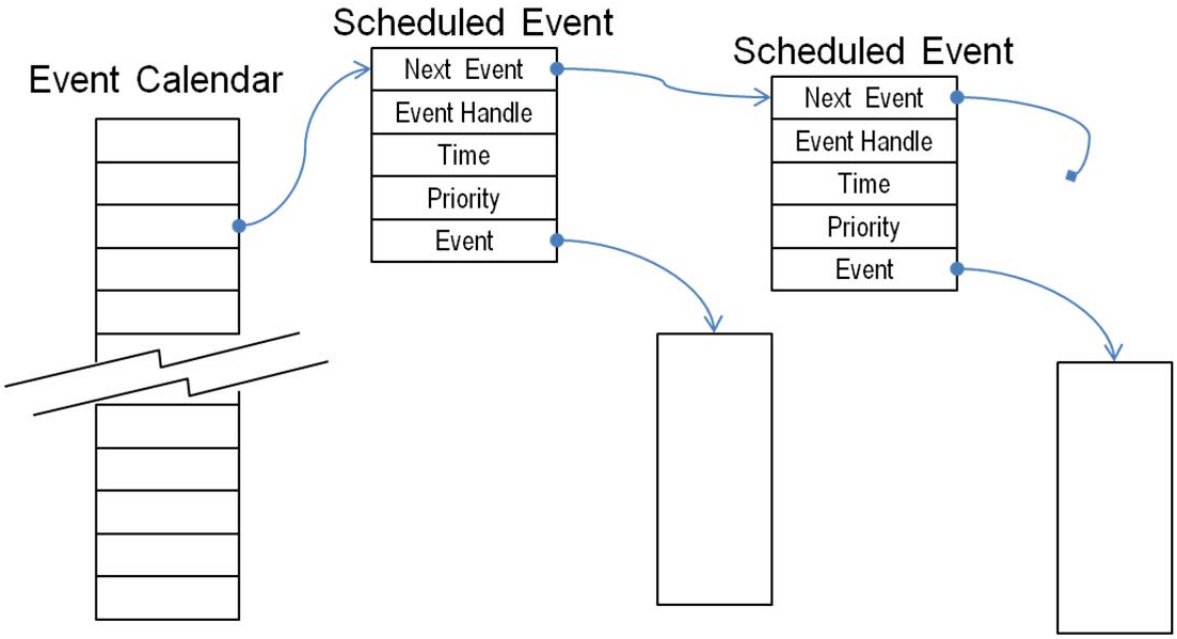

Figure 19 Data Structure for Event Scheduling

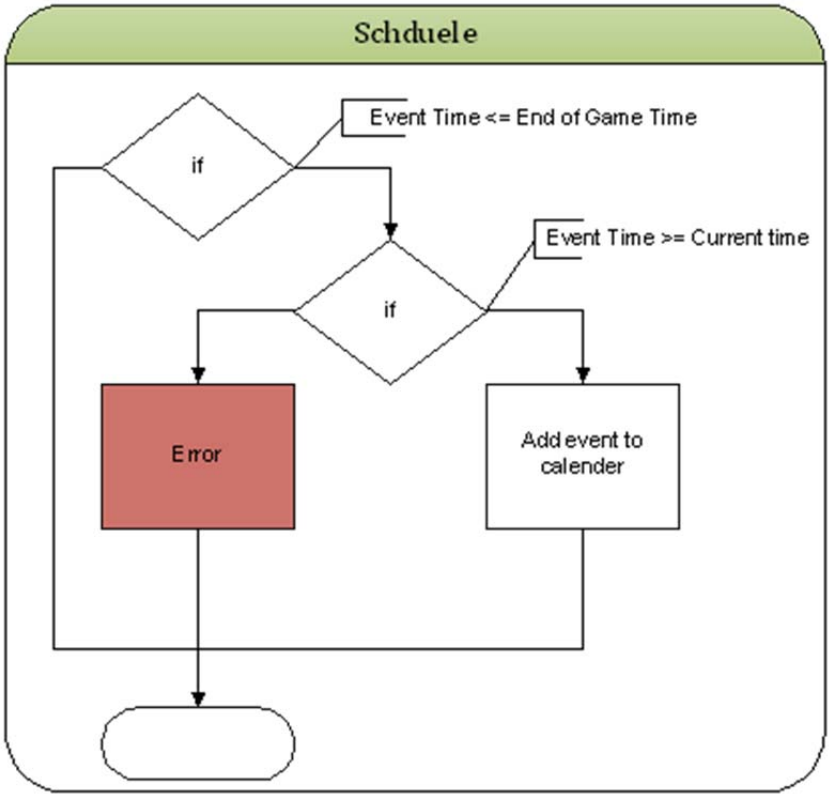

Figure 20 Schedule Workflow

Services

Services are a collection of common tools designed to aid in the development and execution of models. Some examples of Services:

1. Random number generation and distributions
2. Coordinate conversion
3. Unit conversions
4. Math utilities
5. Debugging tools

For this project a limited number of services will be implemented. The number and complexity of the Services implemented will be driven by the requirements of this study.

Current services available: Print Event Calendar, Print Geo Location Table, and Print Entity Table. These services are specific debug utilities; they aid in the debug and simulation verification and validation process.

Services are implemented separately from the simulation engine and in theory can be published/ discovered. However, further study may find limitations on this flexibility.

Common Environment

A Model will interface with the simulation environment through the Simulation Engine (SE). The common environment is a major interface mechanism between models. Through the common environment models will be able to:

1. Expos state variables
2. Post current location
3. Access search routines

When designing a model a paradigm similar to publishing and subscribing to and HLA federation is useful to understand how models interface with the simulation common environment. For example when a model exposes its entity's state to the common environment it is similar to sending an entity object of entity state PDU in a distributed federation.

Currently the common environment consists of the Geo Location Table, the Entity Table, and the current time (more environment tables will be added as the system matures).

Current Time Function

Models can access the current game time with:

int GetCurrentGameTime(void);
void PrintTime(int Type);

Geo Location Table Function

For an entity to be visible to other entities a model must add the entity to the Geo Location Table. A model posts an entity to the Geo Location Table by invoking:

GeoLocation *AddToGeoLocationTable(int X, int Y, int Z, GeoLocation *Loc);

A model can search a geographical area by calling:

EntityList *SearchGeoLocationBox(int MinX, int MinY, int MaxX, int MaxY);

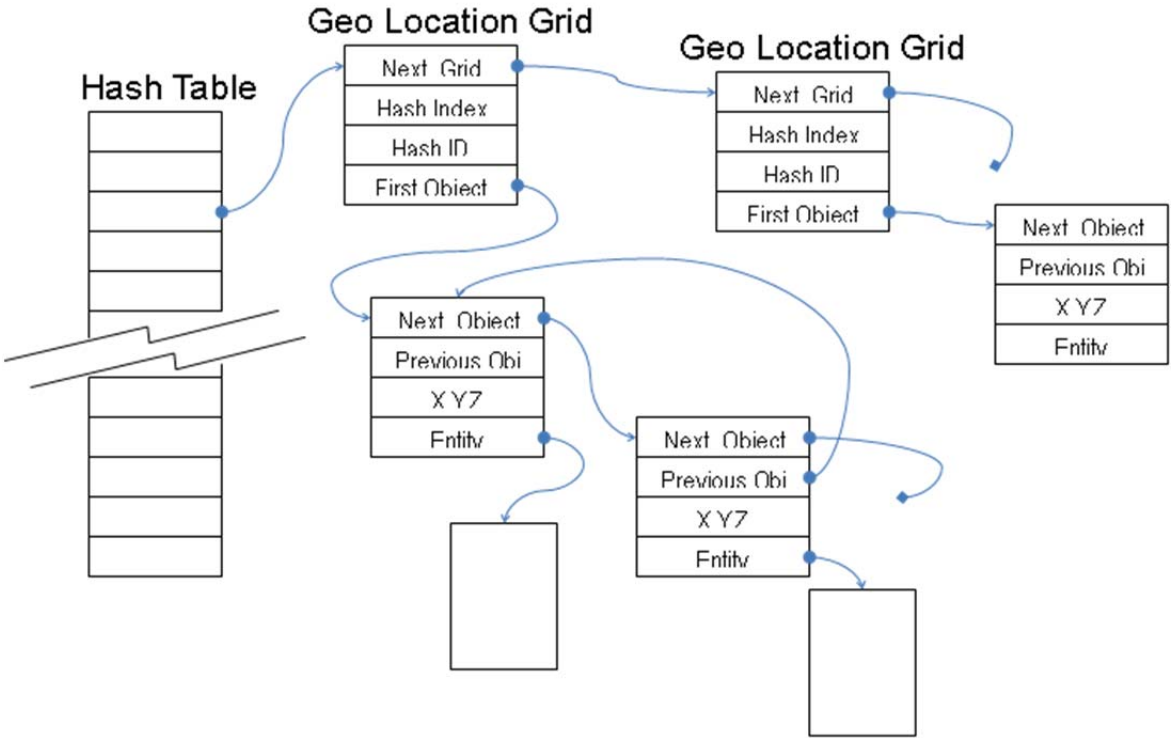

Figure 21 Data Structure for Geo Location

Entity Table Function

For an entities state to be viable to other entities the model must post the entities state to the Entity Table Using:

void SetEntitySide(int EntityID, int In);
void SetEntityLength(int EntityID, int In);
void SetEntityWidth(int EntityID, int In);
void SetEntityHeight(int EntityID, int In) ;
void SetEntityCondition(int EntityID, int In) ;

Currently no model has been implement requiring access to the entity table

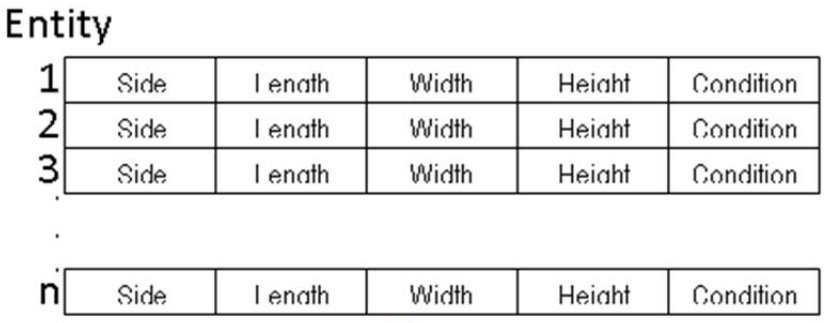

Figure 22 Data Structure for Entity Table

Appendix B – SimAAS Simulation Engine Federated Model Agreement

General

Developers creating modes for inclusion within the Simulation-As-A-Service Simulation Engine must adhere to the process and interfaces described in this document. This is a living document for an immature system and will be updated as needed.

Currently all models must be written to standard c++ compliable on Linux.

Not the SimAAS Simulation Engine and this document are not mature. There are gaps in the martial provide and with time this document will become more useful.

Publishing

For the Composer to include a model for selection by a user a model must publish two set of data: Model Description and Data Description. Currently the publishing/discovering process between a model and the composer is done manually. The required data is entered into a relational database assessable by the Composer.

Model Description

The Model Description database contains eight fields described below.

1. **Name:** an alpha numeric field representing the internal or class name of the model. This name will be used by the auto-code generation part of the composer to build the simulation executable.
2. **Long Name:** a human understandable name for this model. It will be used by the composer to display the model name for user selection.
3. **Type:** an alpha enumeration of the class of model. See below for enumerate list of available model types.
4. **Event Name:** identifies an event to be scheduled by the time keeper. This field is used by the composer to auto generate the code needed to associate this event to its model. This is an alphanumeric string unique to the federation (Note: currently a process to insure the uniqueness of this string dose not exists). If the model will not schedule an event then this field will be set to 'None'.
5. **Event Structure:** identifies the c++ structure name needed to build the code for event scheduling. Note: future implantation of the Simulation Engine may define a universal event structure making this field obsolete. If the event name field is set to 'None" then this field will be set to 'None'.
6. **Required Model:** field indicates if this model required a link to another model type. This field is limited to the same enumeration as the model type field. If there are not any required links this field is set to 'None'.

7. **Optional Model:** indicates if this model can link to another model type. This field is limited to the same enumeration as the model type field. If there are not any required links this field is set to 'None'.
8. **Description:** a user understandable description of the function of the model.

Enumerated Model Types

Currently available model types:

1. Physical
2. Movement
3. Behavior
4. Sensor
5. Data Collection

More model type will be added to the Federated Model Agreement as the environment mutures.

Data Description

The data description documents the date requirement for the model. This data allows the Composer to prompt the user for scenario information.

The Data Description data base record consists of a model name and a series of repeated (0 to n) tuples of input data required.

1. **Model Name:** an alpha numeric field representing the internal or class name of the model. This must match the model name in the Model Description database and is use to link the two databases.
2. **Repeated Tuples:** Repeated from 0 to n required input data. Each tuple contains:
    a. **Field Name:** a short description of the data field.
    b. **Field Type:** currently limited to int, float, and Array. A tuple of type Array indicated the following topple definitions is repeated to the end of the data set. That is if two tuples X and Y follow a tuple of type Array then the reaming data will be interpreted as an array of X, Y, X, Y…
    c. Field Documentation: a user understandable description of the data required for this field. It should contain units and bounds.

Services and Models

There will be a limited number of models implemented for this project. The number and type will be determined by the need to understand the requirements for the Composer. Implemented models will fall in to six general categories (see diagram below): Behavior models, Movement models, Physical models, Sensor models, Debugging services, and Data Logger services.

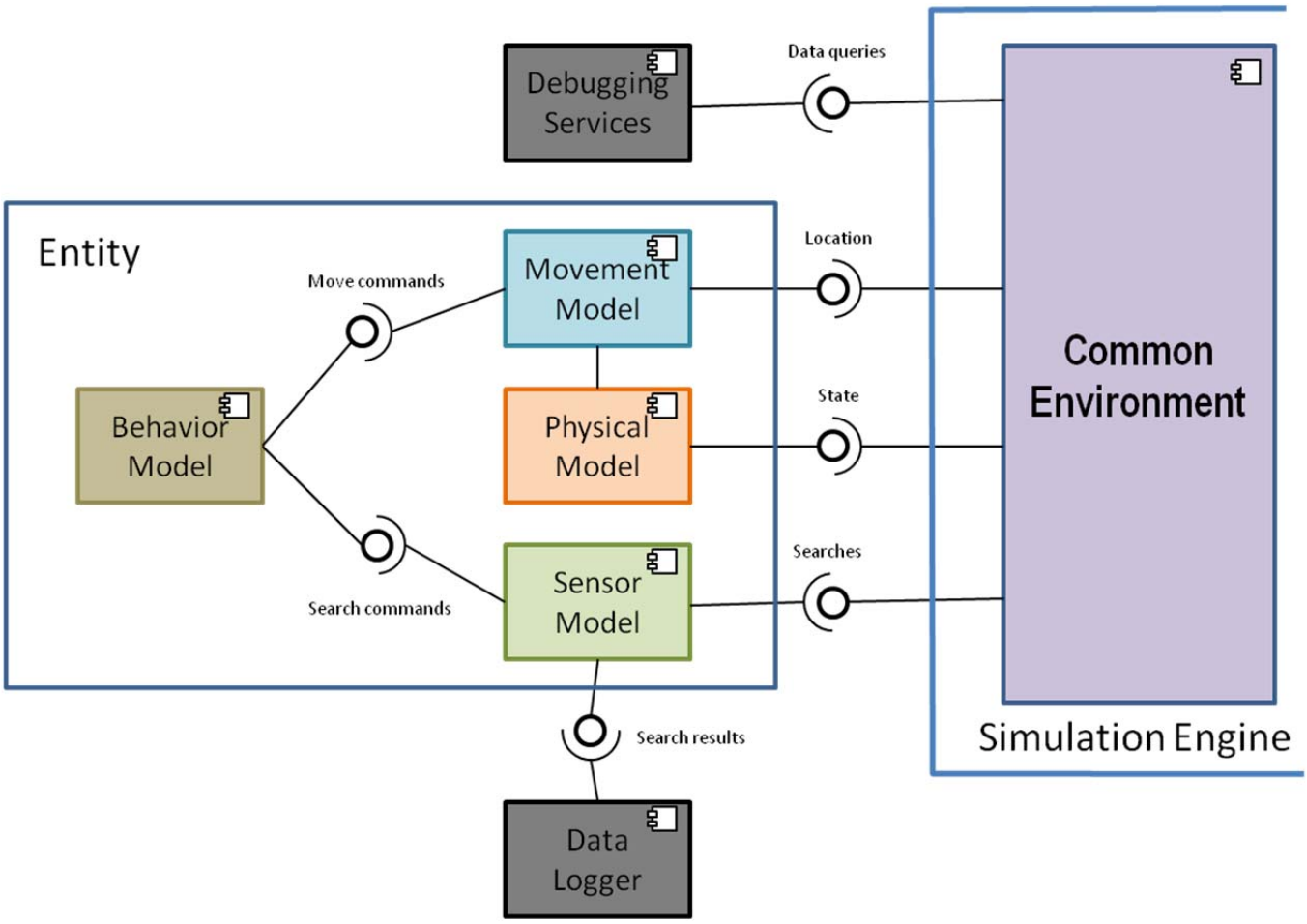

Figure 23 Models Component View Example

Entities will be composed of a selectable group of models. An entity may contain more than one a type of model (such as sensor of behavior models) but will be limited to one physical model and one movement model.

Physical models will provide data and proprieties of the entities state and appearance. Each entity must have one physical model. Physical models must inherit the base Model class. There are no virtual functions required.

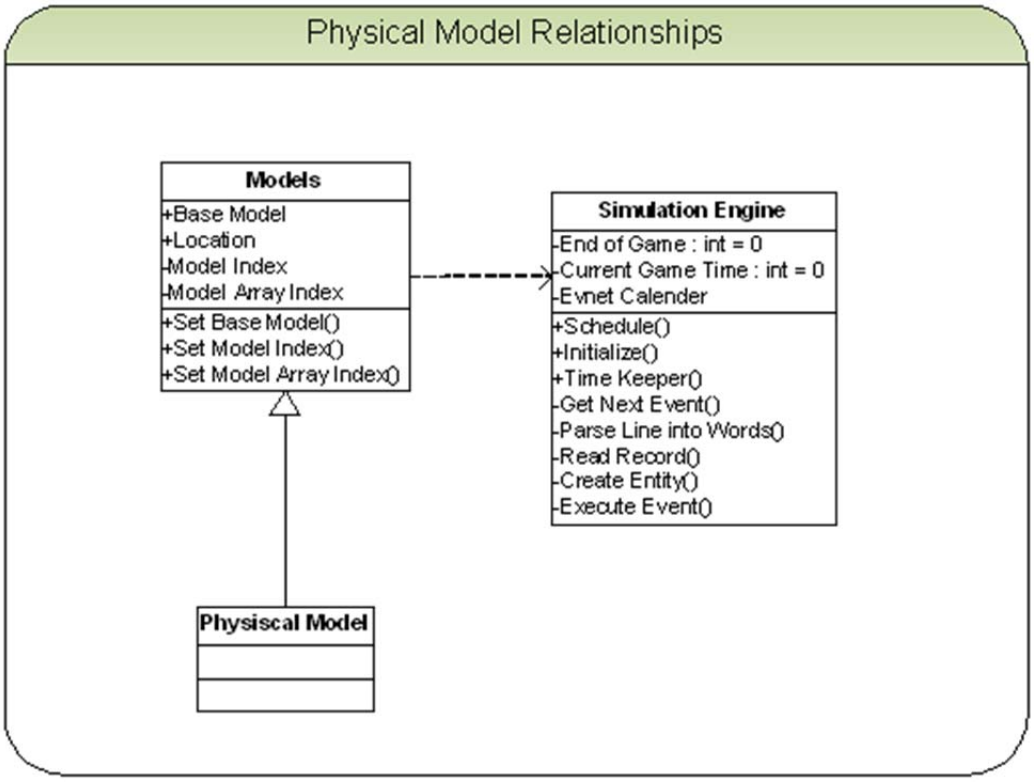

Figure 24 Physical Model Relationships

Movement models will post the current location of the entity into the common environment geo-location data base for other entities to interact with. Movement models must inherit the Movement Model class which inherits the Model base class. Also, they must implement the virtual function *MoveToPoint* in order to interact with their controlling Behavior model.

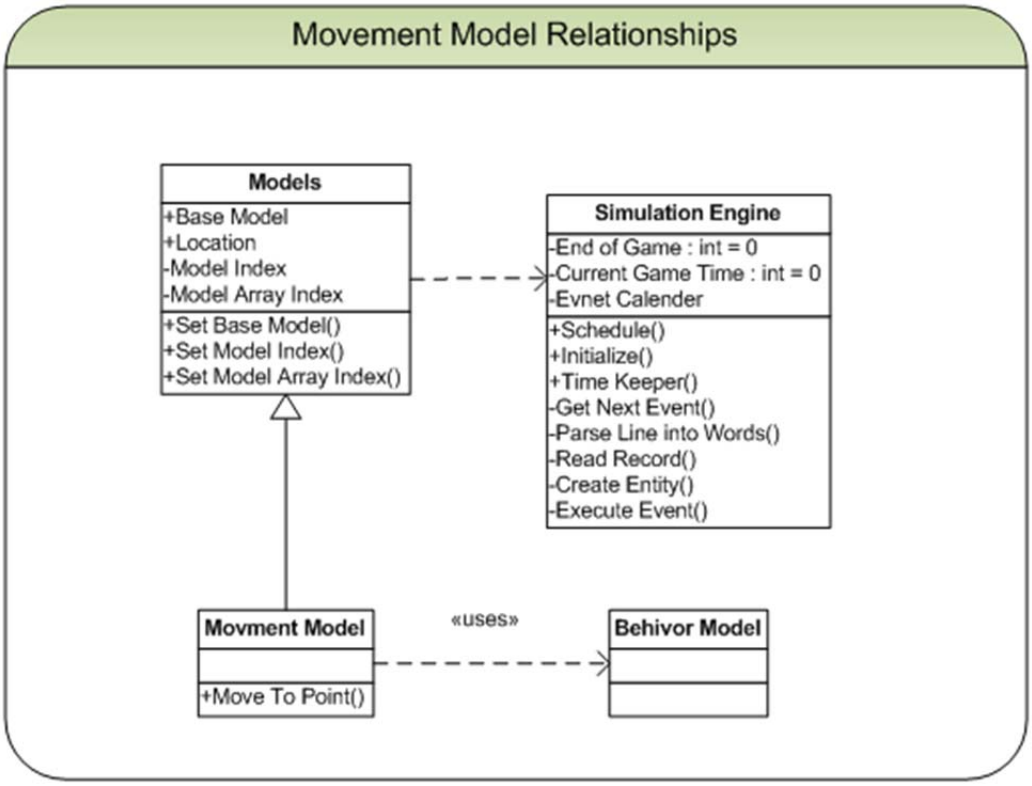

Figure 25 Movement Model Class Relationships

Behavior models control the timing of the ‘when’ and ‘why’ of an entities actions. For example a sensor behavior model would command the sensor model when and where to look in the simulation geo-space. Behavior modes inherit the Behavior Model class which inherits the base Model class. Behavior models must implement the *NextPoint* function to interface with movement models and the *SearchResults* function to interface with sensor models.

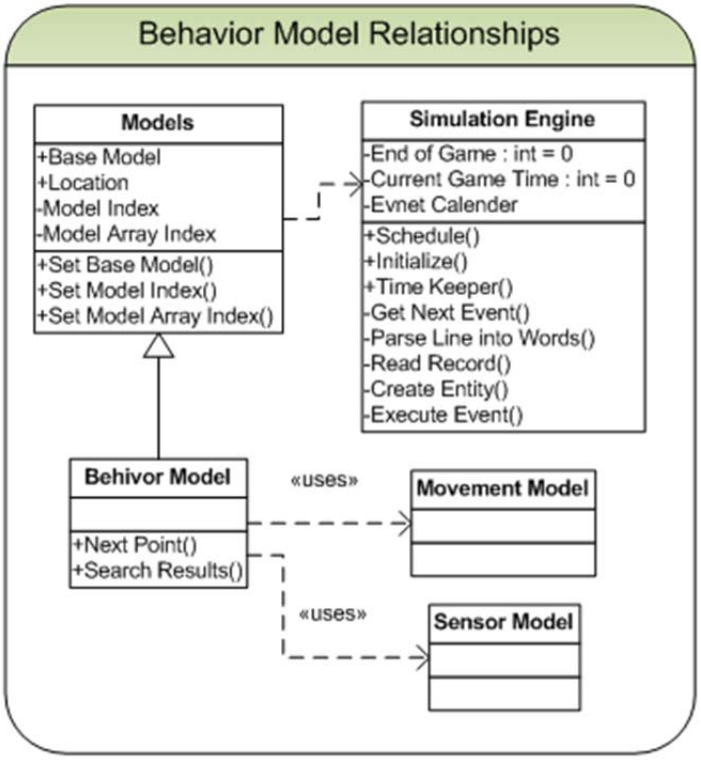

Figure 26 Behavior Model Class Relationships

Sensor models will quire the geo-location data base. Sensor models inherit the Sensor Model class and are required to implement the SearchArea function.

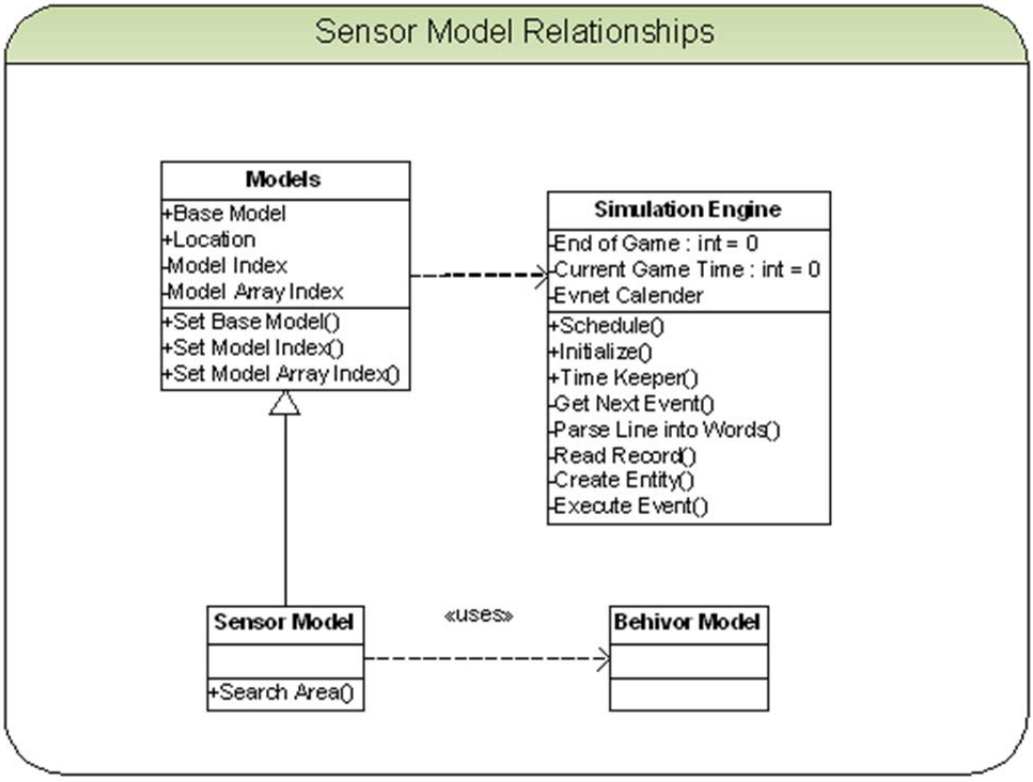

Figure 27 Sensor Model Class Relationships

Data logging services will record specific information based on stimulus form other models. Data loggers inherit the DataCollectionModel class.

Debugging services will record specific information independent of any entity or model. An Entity Location data collecting service will be implemented for this project.

Interactions between Models and the Common Environment

Interactions between the Common Environment and model are a critical component of the simulation. The Common Environment (CE) can be thought of as a distributed interface but local to a specific platform. The information posted to the CE is similar to the information passed with messages such as an Entity State PDU from DIS federates (entity location and state).

Each Model will be associated with the Simulation Engine class. This will give access to the Simulation Engine's member functions (services, environment, and time keeping).

Services are a collection of common tools designed to aid in the development and execution of models. Some examples of Services:

6. Random number generation and distributions
7. Coordinate conversion
8. Unit conversions
9. Math utilities
10. Debugging tools

A Model will interface with the simulation environment through the Simulation Engine (SE). The common environment is a major interface mechanism between models. Through the common environment models will be able to:

4. Expos state variables
5. Post current location
6. Access search routines

Currently the common environment consists of the Geo Location Table, the Entity Table, and the current time (more environment tables will be added as the system matures).

Geo Location Table Function

Non-null movement models must post initial location to the Geo Location Table. This allows an entity to be visible to other entities. The simulation engine uses a Cartesian X, Y, Z coordinate system. A model posts an entity's location to the Geo Location Table by invoking with the *GeoLocation* Loc pointer zero:

GeoLocation *AddToGeoLocationTable(int X, int Y, int Z, GeoLocation *Loc);

AddToGeolocationTable will return the *GeoLocation* pointer to be used latter to update the entity's location.

A movement model updates the coordinate by calling *AddtoGeoLocationTable* with its *GeoLocation* pointer.

A sensor model can search a geographical area by calling:

EntityList *SearchGeoLocationBox(int MinX, int MinY, int MaxX, int MaxY);

The returned *EntitList* is a link list of all entities within the search box specified. Any additional filtering must be done by the sensor model according to the type of sensor modeled.

Entity Table Function

Non-null physical models must post a minimum amount of state information. For an entities state to be viable to other entities the model must post the entities state to the Entity Table Using:

void SetEntitySide(int EntityID, int In);
void SetEntityLength(int EntityID, int In);
void SetEntityWidth(int EntityID, int In);
void SetEntityHeight(int EntityID, int In) ;
void SetEntityCondition(int EntityID, int In) ;

Currently no model has been implement requiring access to the entity table. Additional state variables will be added as the simulation matures.

Interactions between Models and the Time Keeper

Scheduling

When a model schedules an event it will do so through the SE Scheduling service. A model must provide a virtual function called *Event* in order to schedule events. If a model needs to schedule more than one event then it must implement a sub event system and handle the different event internally. See the simulation header file for more details.

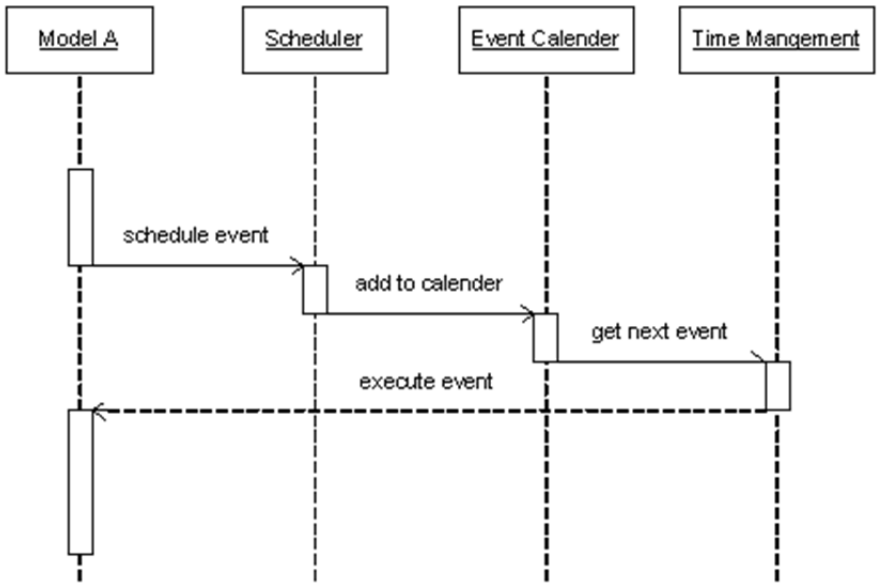

Figure 28 Event Scheduling Sequence Diagram

Scheduling an event is done by a call to the Simulation Engine member function Schedule. The function:

SimulationEngine:: Schedule(int EventIndex, int Time, int Priority, Void *Event)

Where:

EventIndex is a unique identifier for the scheduled model. This is the same string published in the Model Description database. The composer will add this sting to the *EventIndexs* enumeration record created at code auto generation time. This enumeration list can be found in the composer generated file *include/Models.AG.hh*.

Time is when this event will occur (in game time units). This time must be in the future or the Scheduler will drop this event.

Priority is the importance of the event relative to other events occurring at the same time. This is a tie breaker and if two events with the same time and priority are scheduled then the first one scheduled it the first to execute. The legal values are:

0: High

1: Normal

2: Low

There is the enumerated field *EVENT_PRIORITY* located in the SE header file.

Event is a pointer to a block of data required by the scheduled model to function. The structure of the event is unknown to the scheduler.

Current Time Function

Time is maintained by the Time Keeper and is in integer seconds. Models can access the current game time with:

```
int GetCurrentGameTime(void);
void PrintTime(int Type);
```

Intra-Model interactions

Models interact with each other through two methods: Event scheduling and model type specific interactions.

Event scheduling is where one model schedules another through the simulation engine scheduler. This requires a understanding between model not covered within this document (and may not be properly handled by the composer).

The other methods of intra model interactions are a series of model type specific behaviors. These exchanges can be like the Interactions in a HLA environment. Currently there are two sets of interaction defined. More interactions will be added as the environment matures.

Behaviors and Movement Models

Behavior models tell Movement models where to go with the virtual function *MoveToPoint*. Movement models inform their Behavior models when they have reached the target point with the *NextPoint* virtual function.

Behaviors and Sensor Models

Behavior models request a Sensor model to search an area with the *SearchArea* virtual function. Sensor models respond with search results using the *SearchResults* function.

Initialization Requirements

The initialization case will read the Initialization file created by the composer to:

1. Register entity with the common environment.
2. Initialize the base state of each entity.
3. Schedule the first event of each entity.

Appendix C - Initialization file Structure

Overview

This document describes the data structure for the Composer generated simulation initialization file. This file is text based with a string keyword indicating the data structure. Below is a description of each keyword, how to define an entity, and an example.

Keywords

There are a limited number of keywords defined for the initialization file. Each line begins with a text keyword. The keywords are case sensitive.

EndOfGame

The keyword ***EndOfGame*** specifies, in seconds, the length of the simulation run.

Field	Type	Description
EndOfGame	keyword	
Time	integer	Length of simulation run in seconds. Note: this is simulated time not real time. The simulation will run as fast as the CPU will allow.

Entity

The keyword ***Entity*** is used to create each entity within the scenario. A detailed description of how to create an entity is below.The next field will be the entities name and the last field will be the number of non-required models composing this entity. Example: ***Enitiy,Truck1,2***

Field	Type	Description
Entity	keyword	
Entity Name	string	Unique name of this entity
Number of Non-required Models	integer	Number of models needed by this entity beyond the two required.

Physical

The keyword ***Physical*** follows immediately after the ***Entity*** keyword. This specifies the physical model for this entity. This is a required model.

Field	Type	Description
Physical	keyword	
Model Index	Enumeration	Enumerated ID for model. This will be found in the auto-generated file AutoGen/include/ModelIndex.AG.hh

Data[]		Additional data specific to this model. This data will correspond to the model's publish data requirements

Movement

The keyword ***Movement*** follows immediately after the ***Physical*** keyword record. This specifies the movement model for this entity. This is the second required model

Field	Type	Description
Movement	keyword	
Model Index	Enumeration	Enumerated ID for model. This will be found in the auto-generated file AutoGen/include/ModelIndex.AG.hh
Data[]		Additional data specific to this model. This data will correspond to the model's publish data requirements

Behavior and Sensor

The keyword ***Behavior*** or ***Sensor*** specifies an optional model for this entity. The number of optional models has to correspond to the '*number of non-required models*' on the Entity record.

Field	Type	Description
Behavior or **Sensor**	keyword	
Model Index	Enumeration	Enumerated ID for model. This will be found in the auto-generated file AutoGen/include/ModelIndex.AG.hh
Data[]		Additional data specific to this model. This data will correspond to the model's publish data requirements

END

The keyword ***END*** completes the list of models for a given entity. There is no data for this keyword

DataCollection

The ***DataCollection*** keyword is used to instantiates a data collection services.

Field	Type	Description
DataCollection	keyword	
Model Index	Enumeration	Enumerated ID for model. This will be found in the auto-generated file AutoGen/include/ModelIndex.AG.hh
Data[]		Additional data specific to this model. This data will correspond to the model's publish data requirements

Entity Creation

Data describing an entity is defined between the keywords ***Entity*** and ***END*.** The entity must have a physical and a movement model (in that order). Behavior and Sensor models are optional. The below is an example of an entity named Yankee 1 with only the required two models.

- **Entity, Yankee 1, 0**
- **Physical, 3, 1.2, Red…**
- **Movement, 6, 120 …**
- **END**

Example File

The below is an example of an initialization file. This example has seven entities and one data collection service. This simulation will run for 100 seconds.

```
EndOfGame, 100
DataCollection, 4
Entity, Off Map, 0
Physical, 10
Movement, 7
END
Entity, Stationary, 0
Physical, 9, 0, 1, 1, 2
Movement, 8, 12340, 23450
END
Entity, Ground Mover, 1
Physical, 11, 0, 50, 50, 100
Movement, 6, 11340, 22450, 3, 2
Behavior, 2, 3, 11360, 22450, 11360, 22470, 11380, 22470
END
Entity, Ground Mover 2, 1
Physical, 9, 0, 50, 50, 100
Movement, 6, 11340, 22450, 3, 2
Behavior, 2, 3, 12360, 23450, 12360, 23470, 11380, 22470
END
Entity, Air Mover, 0
Physical, 11, 1, 50, 50, 100
Movement, 5, 13000, 22000, 3, 2, 2
Behavior, 1, 2, 13000, 22500, 100, 13000, 23000, 200
END
Entity, Ground Sensor, 2
Physical, 11, 0, 25, 25, 75
Movement, 8, 11300, 22400
Behavior, 3, 12340, 23450, 13, 4
Sensor, 13, 1000
END
Entity, Air Sensor, 2
Physical, 11, 0, 25, 25, 75
Movement, 8, 11300, 22400
Behavior, 3, 13000, 22000, 12, 4
Sensor, 12, 1000
END
```