



**VISUAL UNIFIED MODELING LANGUAGE FOR THE COMPOSITION OF
SCENARIOS IN MODELING AND SIMULATION SYSTEMS**

THESIS

Daniel E. Swayne, Master Sergeant, USAF

AFIT/GCS/ENG/04-19

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GCS/ENG/04-19

**VISUAL UNIFIED MODELING LANGUAGE FOR THE COMPOSITION OF
SCENARIOS IN MODELING AND SIMULATION SYSTEMS**

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Daniel E. Swayne, BS

Master Sergeant, USAF

August 2004

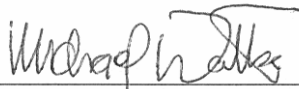
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**VISUAL UNIFIED MODELING LANGUAGE FOR THE COMPOSITION OF
SCENARIOS IN MODELING AND SIMULATION SYSTEMS**

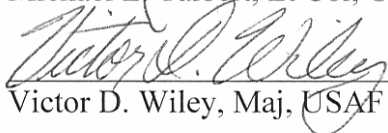
Daniel E. Swayne, BS

Master Sergeant, USAF

Approved:



Michael L. Talbert, Lt Col, USAF (Chairman)



Victor D. Wiley, Maj, USAF (Member)



Henry B. Potoczny, Ph.D., AFIT (Member)

27 Aug 04
Date

27 Aug 04
Date

August 27, 2004
Date

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Lt Col Michael L. Talbert, for his guidance and support throughout the course of this thesis effort. His expertise in the area of research provided me views into many facets of the subject that I was previously unaware. His encouragement and assistance made this work possible.

I would also like to express gratitude and thanks to my wife of over twenty years for her unflagging support and encouragement. She graciously accepted the reduced level of “quality time” with me and the increased level of “child interception” in order to provide me with sufficient time and resources to complete this effort. Along with my wife, I would like to thank my children for putting up with a “ghost father,” especially when they would rather play games or wrassle with me.

I would like to thank the Secretary of the Air Force, James Roche, for creating the opportunity in allowing this first class of Air Force enlisted to attend the Air Force Institute of Technology. I likewise would like to thank SMSgt Booker for asking the question that started the ball rolling in this direction.

Finally, I would like to thank my Lord and Savior for His grace to carry me through this intense period of life. His grace is clearly sufficient and abundant.

Daniel E. Swayne

Table of Contents

	Page
Acknowledgments.....	iv
Table of Contents.....	v
List of Figures	ix
List of Tables	xi
Abstract	xii
I. Introduction	1
1.1 Background	1
1.2 Problem Statement	3
1.3 Research Objectives	4
1.3.1 Methodology	5
1.3.2 Assumptions/Limitations	6
1.3.3 Implications.....	6
1.4 Preview.....	7
II. Literature Review	9
2.1 Introduction	9
2.2 Simulation Systems	9
2.3 Research at AFIT	11
2.3.1 Scenario Reuse Research	12
2.3.2 Visual Language Research.....	15
2.4 Other Research and Development.....	19
2.4.1 Modular Interoperable Synthetic Environment (ModISE)	20
2.4.2 SimBionic	21

2.4.3	Analysis and Description of Requirements and Architecture (ADORA).....	24
2.4.3.1	Description and Analysis of ADORA	25
2.4.3.2	Conclusion on ADORA.....	28
2.5	Summary	28
III.	Methodology	30
3.1	Introduction	30
3.2	Background	30
3.2.1	Semantic Meaning in Existing Scenario Components.....	30
3.2.2	Unified Visual Language	33
3.3	Design Principles.....	35
3.4	Evaluation Criteria	37
3.5	Summary	38
IV.	Conceptual Design of the Visual Language	39
4.1	Introduction	39
4.2	Abstract Entity Structure.....	40
4.2.1	Abstract Entity Attributes and Entity Types.....	41
4.2.2	Abstract Entity and Abstract Entity Set.....	43
4.2.3	Nested Abstract Entities.....	45
4.2.4	Transitions and Abstract Entities	47
4.2.5	Abstract Entities and Behavior Maps	48
4.3	Entity Behavior	50
4.3.1	Nodes	50
4.3.1.1	Multitask Nodes.....	51

4.3.1.2	Nodes and Attributes	53
4.3.2	Transitions.....	54
4.3.2.1	Conditional Transitions	54
4.3.2.2.	Reaction Transitions.....	56
4.4	Concrete Entity Binding.....	58
4.5	Summary	59
V.	Implementation and Use Cases of the Visual Language.....	61
5.1	Introduction	61
5.2	Implementation of the Visual Language Tool.....	61
5.2.1	Graphical User Interface Component	63
5.2.1.1	Creating Entity Types and Subtypes	66
5.2.1.2	Creating Entities	70
5.2.1.3	Creating Entity Transitions.....	72
5.2.1.4	Creating Behaviors	72
5.2.1.4	Reusing Abstract Entities and Behaviors	75
5.2.1.5	Converting the Abstract Scenario to Concrete Scenario	75
5.2.2	Storage and Retrieval of Data Components.....	76
5.3	Example Abstract Scenario Design in the Language Tool	77
5.4	Summary	79
VI.	Conclusions and Recommendations	81
6.1	Introduction	81
6.2	Conclusions of Research	81
6.3	Significance of Research.....	82

6.4	Recommendations for Action.....	83
6.5	Recommendations for Future Research	83
6.6	Summary	84
Appendix A.....		85
Appendix B.....		86
Appendix C		90
Bibliography		93
Vita		96

List of Figures

	Page
Figure 1. AFIT Research in Scenario Reuse and Visual Tools	3
Figure 2. Hierarchy of simulation types [10:782].....	10
Figure 3. Global Object-Oriented Database for Simulation Systems [6:4]	13
Figure 4. Elements of the Scenario-Based Specification Diagram (SBSD) [2:79, 85] ...	16
Figure 5. An SBSD diagram with a Multitask Node [2:82]	17
Figure 6. An SBSD diagram with a Temporary Reaction Transition [2:93].....	18
Figure 7. A Treemap of a OneSAF Scenario [2:114]	19
Figure 8. Overview of the SimBionic system [16:16]	22
Figure 9. Example of SimBionic behavior graph [7:82].....	23
Figure 10. An ADORA base view [9:427]	27
Figure 11. A partitioned ADORA base view (of the RoomModule abstract object set) combined with its behavior [9:432]	27
Figure 12. Generation of a Scenario [6:37]	31
Figure 13. Abstract entity (left) and abstract entity set (right)	40
Figure 14. Abstract object-oriented view versus class-oriented view [9:427].....	41
Figure 15. Nested abstract entities	46
Figure 16. Communication and Reaction Transitions between Entities.....	47
Figure 17. Behavior Map Status of Abstract Entities	49
Figure 18. Atomic and Multitask Nodes.....	53
Figure 19. Conditional Transition between two Atomic Nodes	55

Figure 20. Permanent and Temporary Reactions.....	56
Figure 21. UML Class Diagram of the Visual Tool Program GUI Component.....	62
Figure 22. Visual Scenario Language Tool Screenshot.....	63
Figure 23. Entity Type editor panel in VSL tool	67
Figure 24. Simple Scenario in VSL	78
Figure 25. Detailed View of Fly Route Task Frame.....	79
Figure 26. Personal correspondence from the Defense Modeling and Simulation Office (DSMO), May 2003	85

List of Tables

	Page
Table 1. Methodology for Tool Integration [13:32]	14
Table 2. Visual Language in BNF (basic definitions)	86
Table 3. Visual Language in BNF (abstract entity definitions)	87
Table 4. Visual Language in BNF (behavior definitions).....	88

Abstract

The Department of Defense uses modeling and simulation systems in many various roles, from research and training to modeling likely outcomes of command decisions. Simulation systems have been increasing in complexity with the increased capability of low-cost computer systems to support these DOD requirements. The demand for scenarios is also increasing, but the complexity of the simulation systems has caused a bottleneck in scenario development due to the limited number of individuals with knowledge of the arcane simulator languages in which these scenarios are written.

This research combines the results of previous AFIT efforts in visual modeling languages to create a language that unifies description of entities within a scenario with its behavior using a visual tool that was developed in the course of this research. The resulting language has a grammar and syntax that can be parsed from the visual representation of the scenario. The language is designed so that scenarios can be described in a generic manner, not tied to a specific simulation system, allowing the future development of modules to translate the generic scenario into simulation system specific scenarios.

VISUAL UNIFIED MODELING LANGUAGE FOR THE COMPOSITION OF SCENARIOS IN MODELING AND SIMULATION SYSTEMS

I. Introduction

The environment that the United States military finds itself in is increasingly complex. Weapon systems today are orders of magnitude greater in sophistication over previous generations. World events also move at a greater speed due to the state of the art in communications equipment, infrastructure and the 24-hour news cycle.

In order to effectively manage resources in this environment, the Department of Defense (DOD) has relied increasingly on simulation and modeling systems. These systems make it possible to design, develop and test system components, concepts, strategic plans, training and deployment—all at a reasonable cost. Nevertheless, the cost of using these simulation systems is also increasing with its own complexity. This cost is primarily in terms of effort and expertise in the construction of scenarios for use within the simulation systems.

1.1 Background

Historically, simulation and modeling systems have “been independent, stand-alone systems that address specific problems and adhere to a unique architecture established by the designer” [17:806]. These independent systems had their own distinct format for entity and scenario depiction due to the architecture and specific analytical focus of the model by the simulation system.

Nevertheless, many entities used were common to the different systems; e.g., one system used F-16 aircraft in its simulation as would another system. These entity models and scenarios of one simulation system were typically incompatible with another. The DOD and other interested parties recognized the utility of interoperable simulation components—entities that can be used across different simulation tools.

Major efforts in achieving interoperability include HLA, the High Level Architecture. This approach provides a framework within which components created apart from a specific simulation modeling system can interact with other components. However, components must comply with HLA requirements to operate in this environment; this means that many excellent legacy entities are not usable within this framework. The question became one of transformation. Could components and scenarios be transformed from their native format into another simulation system's format? Better yet, is it possible to create a universal format that could transform existing components and scenarios into its form and then transform from the universal format into another system format? Ideally, this universal form should support a transform to create scenarios that are HLA compliant.

Past AFIT research on reuse of components of legacy simulation systems have concentrated on database stores and conversions [1; 4; 6; 12; 13]. While this approach directly addresses reuse, construction of scenarios using these components remains a human-intensive effort. Consequently AFIT began research into the use of visual tools to aid in the understanding and ultimately construction of scenarios [5; 2]. This thesis continues this area of research and attempts to bridge the two efforts (Figure 1).

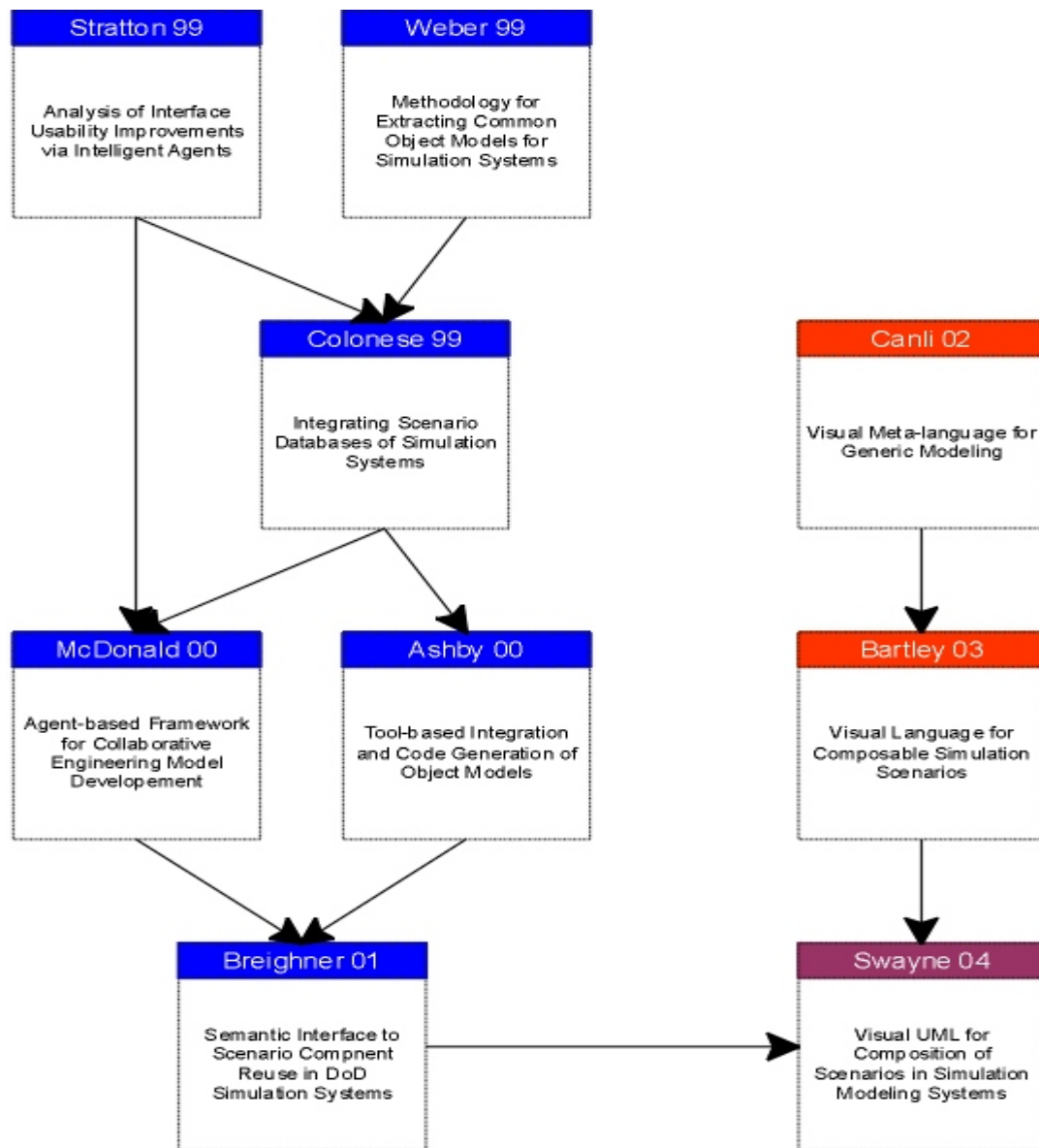


Figure 1. AFIT Research in Scenario Reuse and Visual Tools

1.2 Problem Statement

The preliminary work of Bartley [2] explored the application of graphical description of scenarios, called the Scenario-Based Specification Diagram (SBSD), in an actual simulation modeling system, specifically OneSAF (One Semi-Automated Forces simulator used primarily by the U.S. Army). The work demonstrated the concept as

sound; however, it was limited to describing the scenarios from the system's syntax into a form more readily understood by human users. It did not permit the conversion of visual descriptions into a form that OneSAF could use as input. The present problem then is an extension of Bartley's work—*is a visual language viable that describes scenarios in a generic or universal format that may subsequently be translated into machine usable input for use by specific simulation systems?* The generic format is based on the common object model database format developed by Colonese and refined by Breighner; the details of these efforts are provided in chapter 2.

1.3 Research Objectives

This research focuses primarily on the reduction of visual language components to a granularity fine enough to describe behaviors inherent in a scenario. The key question to the effort is whether it is possible for a visual component language to be rich enough to describe adequately—in terms of precision and without ambiguity—behaviors that can be compiled into a generic or universal form that is ultimately transformed into a machine-usable scenario.

The transformation from generic data form to a specific system form is relatively trivial, provided the generic form contains sufficient detail. Breighner demonstrated this in his research by transforming Suppressor Composite Mission Simulation System (SUPPRESSOR) simulation system scenario components into Simulated Warfare Environment Generator (SWEG) components [4:95-96, 103-111].

The effort here is to determine which semantic elements are required to adequately express the scenario components in the universal (visual) form and then to

implement them in a visual design editor tool. Additionally a translator to a specific simulation system must be created to extract the universal component and transform the scenario component into a form usable to the simulator.

1.3.1 Methodology

This document covers previous AFIT research efforts for extensibility and synthesis as well as the current state of outside research in describing entity behavior models including object-oriented modeling systems such as message sequence charts (MSC) and its variants, hierarchical state machine models and the ADORA modeling language [9; 20]. Many of these modeling systems extend or augment the unified modeling language (UML) and provide a source of insight into what makes an effective behavior descriptor. As scenarios are a marrying of behaviors within a mission to the entities that perform these behaviors, the advancements made in behavior modeling languages will likely contribute to a better understanding of generalized behavior semantics as required for a universal visual simulation scenario editor.

This report also documents the combination of the interoperable scenario database with the visual scenario description language, the merging of two threads of research at AFIT along with the knowledge gleaned from the evaluation of current behavior modeling descriptors. The result is a software tool that allows the user to compose scenarios in a visual manner with a minimum of simulator-specific language knowledge, provided the translation utility exists for the target simulation system.

Finally, this document describes the creation of sample scenarios in the universal visual editor and the generation of the corresponding scenario output for a specific simulation system.

1.3.2 Assumptions/Limitations

It is unlikely that all of the required values for a scenario in a specific simulation system can be provided in a universal tool [6:3]. The tool transforming the scenario from the universal to specific form must provide the simulation system-specific requirements.

1.3.3 Implications

The goal of this research is to demonstrate the viability of the visual scenario language and tools. If successful, these tools will provide the means by which problem domain experts can quickly and reliably create scenarios for simulators. This will reduce the bottleneck of the limited number of software programming experts of a given simulator system—who may or may not understand a given military problem well—to create scenarios for the problem. The effect should be similar to the revolution in software development caused by the progress of computer languages from machine language to modern visual programming environments, such as Visual C++.

Furthermore, the scenarios created with this tool suite should be highly reusable in other simulation systems than the specific system that the author of the scenario may have in mind while creating it. All that would be required is the corresponding transformation tool for the target simulation system.

1.4 Preview

Chapter two provides a summary of a corpus of research at AFIT in the area of simulation component reuse and the current state-of-the-art of work with a component-based integration tool used to create virtual simulations from heterogeneous simulation components (ModISE from Science Application International Corporation), a visual depiction of entity behavior (SimBionic from Stottler Henke) and a highly interesting cohesive object-oriented design charting methodology (ADORA from Institute of Information Science, University of Zürich).

Chapter three discusses the problem of extracting semantic information from the scenario task frames as well as its context in order to store the information in a universal format. This provides the foundation for development of the transformation module to operate with the visual language tool. This semantic information—metadata—must be captured for reuse of existing scenarios into different simulation systems. The metadata, along with specific instance data, must be encoded in the visual tool for end-users to develop universal form scenarios. Additionally, the visual language must be unified—the various attributes or views of the language must work as a coherent whole. This also is discussed in chapter three.

Chapter four documents the implementation of the visual language tool, its database store and the transformation module that permits the creation of specific scenario code usable in a specific simulation system. The architecture of the system is fully defined along with test execution runs.

Chapter five contains the conclusions reached in the effort along with suggestions for further research.

II. Literature Review

2.1 Introduction

There is a very high interest in creating composable simulations: the DOD, other government organizations, industry, and academic communities among others each recognize the value of creating reliable simulations. The data obtained from simulation runs could save an organization literally millions of dollars that would be better directed to other efforts or to provide an even a more efficient tuning of a current activity. The bottleneck, as discussed in the previous chapter, is the growing complexity of simulation systems and the proportionate resources and expertise required to create scenarios in these simulations.

To better understand the current state of research and where best to focus future research, the following review is offered.

2.2 Simulation Systems

Simulation systems vary in focus and scope. From a DOD perspective, simulation types may be described in a hierarchical form (Figure 2) [10:781-782]. The levels, from the bottom of the pyramid, are the specialty level, which focuses on the engineering aspects of a system or system component (electrical, stress, mechanical, etc.); the engagement level, which permits tests of aggregate physical entities (aircraft, ships, missiles, etc.) and may include some level of human interaction; the mission level (tactical-oriented), which begins a higher level of concept and facilitates the development of improved tactics and training; and the campaign level (theater-oriented), which begins

a crossover from tactical towards strategic operations and training. The tip of the pyramid may be considered to represent the next aspect, global strategic, which allows policy makers to ask “what if” questions, and gain insight from the attempt to answer these questions. These insights aid decision makers in developing policy to effect national goals. The focus of this research and review is on the mission level of simulation systems, although the research could be applied to other levels as well.

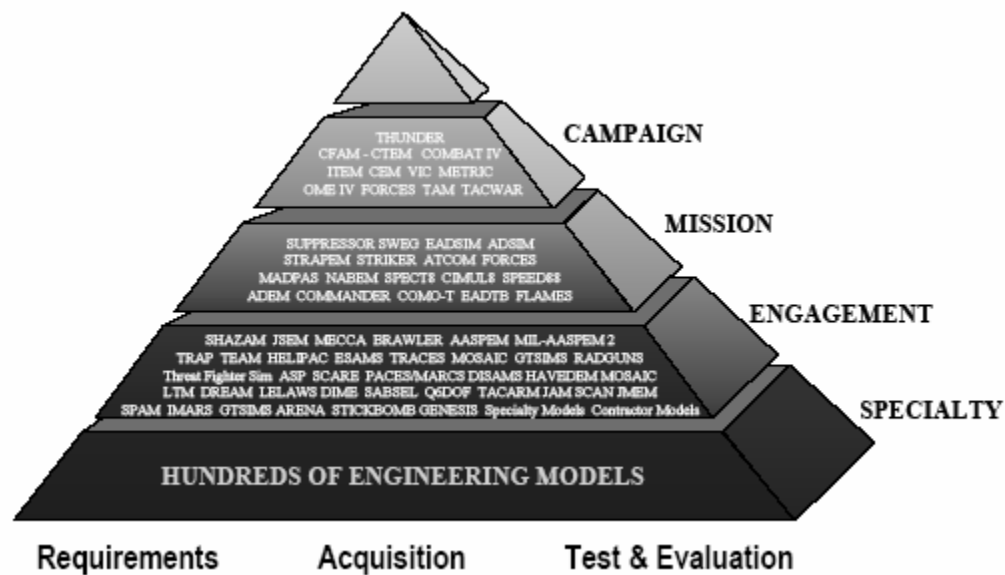


Figure 2. Hierarchy of simulation types [10:782]

The mission level involves components such as aircraft, sensors, terrain, etc. Simulators of this category allow planners to determine if a particular tactic is effective in a specific environment. It also covers simulations that interact with humans in a training situation; trainees could be pilots, operational commanders, tank commanders or any other individual or team member playing out a scenario. There is some overlap with the

lower level of simulation as well, since, for example, a pilot must operate an aircraft that has certain capabilities and constraints.

As noted earlier, these simulation systems are increasingly complex as they must accurately model—to the level required by the scenario—the aircraft, radar and other components with which the simulation user (trainee, tactician) must operate, encounter or otherwise interact. The increased complexity make the simulation systems even more attractive to users as they can press the limits to determine what really can work without actually endangering real people or equipment—or use equipment that has not yet been created. In this way, planners and developers can determine whether the proposed equipment would be worth pursuing.

With the increased complexity comes increased cost to accurately develop scenarios. But because of the widespread interest in the utility of simulations and the recognition of the increasing cost—in time as well as money—of creating scenarios, there has been a large body of work in improving the reusability of simulation components and simplifying the creation of entity behavior, accomplished in the simulation community—private and public organizations as well as academic institutions such as AFIT. The next section highlights some of the relevant research conducted at AFIT.

2.3 Research at AFIT

Relevant research at AFIT has followed generally two threads: one thread has focused on scenario research in terms of capturing data from existing simulation systems

for reuse in other simulation systems; the other thread has focused on visual depictions of components and behavior.

2.3.1 Scenario Reuse Research

Research in the area of scenario reuse began with Weber [19] and Stratton [18]. Weber focused on the extraction of scenario data from existing simulation data stores for the Air Force Research Laboratory (AFRL) into an object-oriented scenario database. Stratton created an interface for users to generate a new scenario instance from the object-oriented scenario database using translator modules for the target simulation system, the Suppressor Composite Mission Simulation System (SUPPRESSOR).

Colonese [6] adapted the framework from Stratton's research and further refined it into a more generalized form. In her research, she described a methodology to integrate data of scenarios from the Extended Air Defense Simulation (EADSIM) and SUPPRESSOR systems using an object-oriented database system. Her work focused on creating a global object-oriented database that could generate scenarios from the global data into the desired target simulation form. Figure 3 shows a general view of Colonese's global database system as it fits with various simulation systems used at AFRL.

Noe [13], while not specifically involved in the reuse of scenarios in disparate simulation systems, focused on fitting software tools to work cooperatively, particularly those tools that were not specifically designed to work as part of an integrated suite. Her research demonstrated that it is possible to create "wrappers" around software such that the net result was an "extension" of capability. What one individual tool could not do, another provided. Noe developed a process to functionally integrate these separate

systems, shown in Table 1. Her methodology allows developers to leverage existing extensibility in the individual systems. The application to simulation systems is the possibility of using components from separate simulation systems in a cooperative manner. For example, one simulation may have the precise implementation of an aircraft, another system a sensor array. Using the principles Noe discovered in her research, it may be possible to put wrappers around the two simulation systems in such a way that the two behave as one system.

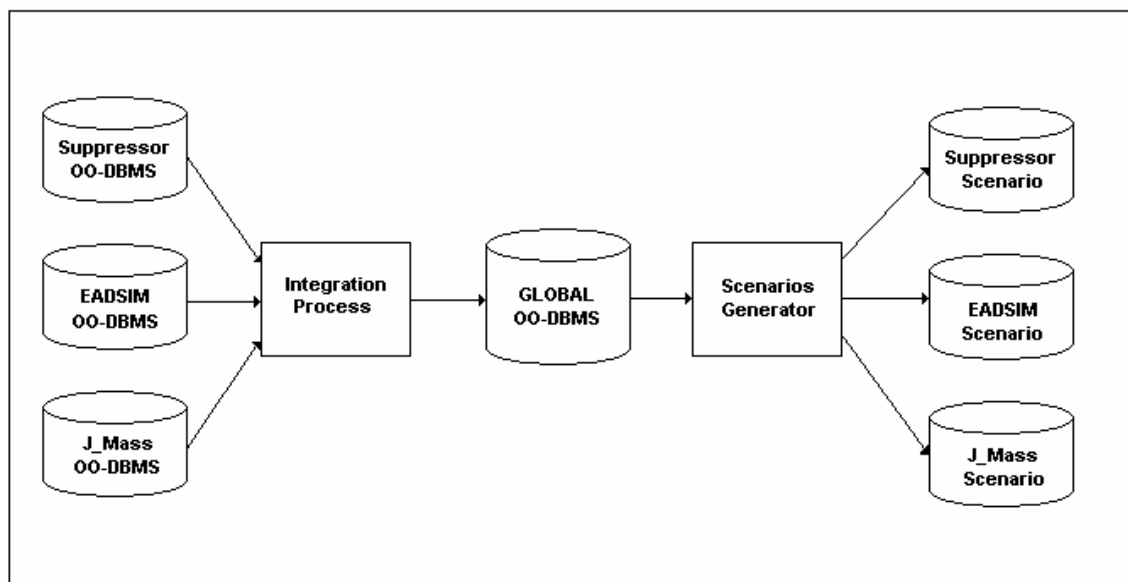


Figure 3. Global Object-Oriented Database for Simulation Systems [6:4]

By using this integration process, it is possible to reuse components. This approach however, still leads to a manual effort to create new scenarios. Nevertheless, it may be possible to automate some of this integration with the cost of some overhead to the original wrapper.

Ashby [1] continued this thread of research by proposing an automated tool to support the global format object-oriented database, a logical extension of the Colonese research, and blending in some of the results of Noe's work.

Table 1. Methodology for Tool Integration [13:32]

Step 1:	Determine for each tool: Input Mechanism Output Mechanism Extendability
Step 2:	Determine for each tool pair: Extendability Class Data Compatibility
Step 3:	Apply design rules to determine structure of system. Provide output of first tool and input of second tool to determine communication path. Apply design rules based on extendability class to determine control integration implementation and data transformation.

McDonald [12] and Breighner [4] concentrated on agent-based searching, retrieval and translation of simulation components. Breighner's work, in particular, focused on rapid selection and retrieval of simulation components. The data stores of the AFRL simulation systems are huge and earlier attempts to create the global object-oriented database still required a significant amount of human intervention, specifically in identifying the semantic essence of a simulation component. Breighner's research produced a system that eased the human intervention required to create and use a catalog of scenario components to construct new scenarios across disparate simulator systems. Additionally, Breighner developed a translator tool to convert SUPPRESSOR scenario components into Simulated Warfare Environment Generator (SWEG) simulation format [4:103-111].

2.3.2 Visual Language Research

Canli [5] began the visual thread of research at AFIT with his work on a visual meta-data language. The language he invented, the Visual Language for Generic Modeling (VLGM), clearly permits rapid construction of entities from simpler components. While his work concentrated on component structure rather than behavior, he was able to demonstrate that UML is inadequate to the task of coherently mapping behavior to scenario components. He also demonstrated the feasibility of visual languages to provide the ability to confirm valid connections between components and conversely reject invalid connections through examples involving electrical components.

Bartley [2] began research at AFIT in modeling behavior of simulation components within a scenario. She first described the requirements of a scenario behavior language—it must allow for reactions, must accept parameters, must represent temporal conditions, must be composable, must focus on activities rather than transitions, and must allow users to define behavior at a high level of abstraction—and then examined available means of describing behavior.

Her research covered traditional diagrams used in software development to describe behavior. While there are many similarities between object-oriented software development and simulation scenario development, there are significant differences. She identified these and demonstrated how the classic diagrams failed to completely fulfill the requirements for a scenario description language, while retaining those aspects that did meet the requirements. As a result, Bartley developed a new behavior diagram language,

forged from these classic sources but with new aspects that allowed the unique requirements for scenario behavior description.

This new visual language tool is called the Scenario-Based Specification Diagram (SBSD) [2:77]. Figure 4 shows the key elements of the SBSD language. SBSD uses nodes to describe tasks that a scenario component is to perform. Upon completion of a task, the behavior chart plots the transition to the next task the component is to perform. This is different from classic approaches in that these task nodes are not simple states that an entity may go through. Indeed, unlike many of the classic charts Bartley discussed (e.g., activity graphs), tasks in SBSD may be interrupted by the arrival of some other event or entity, which she termed a *reaction*-type transition. Otherwise, the activities of a task node must be completed before a transition occurs. Therefore the focus of the diagram is on the activities that a component must perform, rather than the change in data within an object. Tasks may be grouped into a *multitask* node to aid user understanding by abstracting some of the details. The multitask nodes may also be expanded to show the details when required by the system user. Figure 5 shows a multitask node.

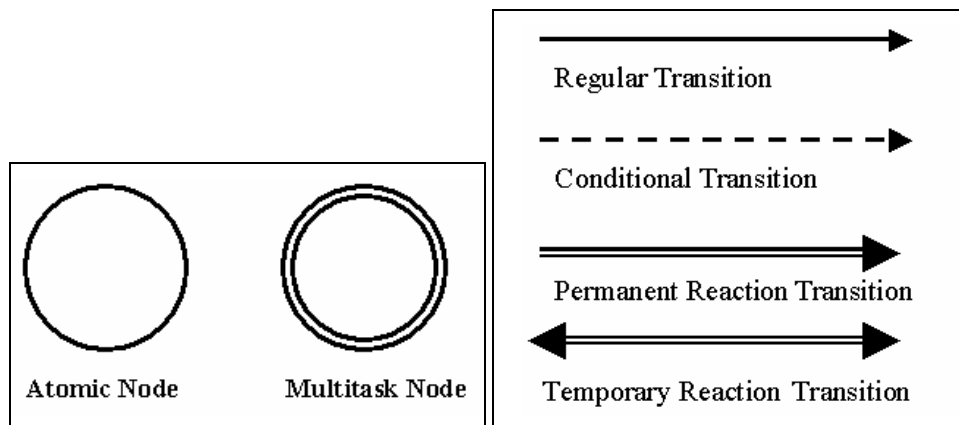


Figure 4. Elements of the Scenario-Based Specification Diagram (SBSD) [2:79, 85]

The requirements of conditions, temporal conditions and reaction events are associated with their corresponding transition. An example containing conditional and temporary reaction transitions with the conditions are shown in Figure 6. In the figure, the aircraft remains at the starting task (awaiting orders) until the associated condition “On Order Command” is satisfied, at which point the aircraft executes task FWA Ingress. Once the aircraft is within 200 feet of the target, the aircraft reacts by executing the Air Attack task. This reaction is permanent; the aircraft does not resume the FWA Ingress upon completion of the Air Attack task.

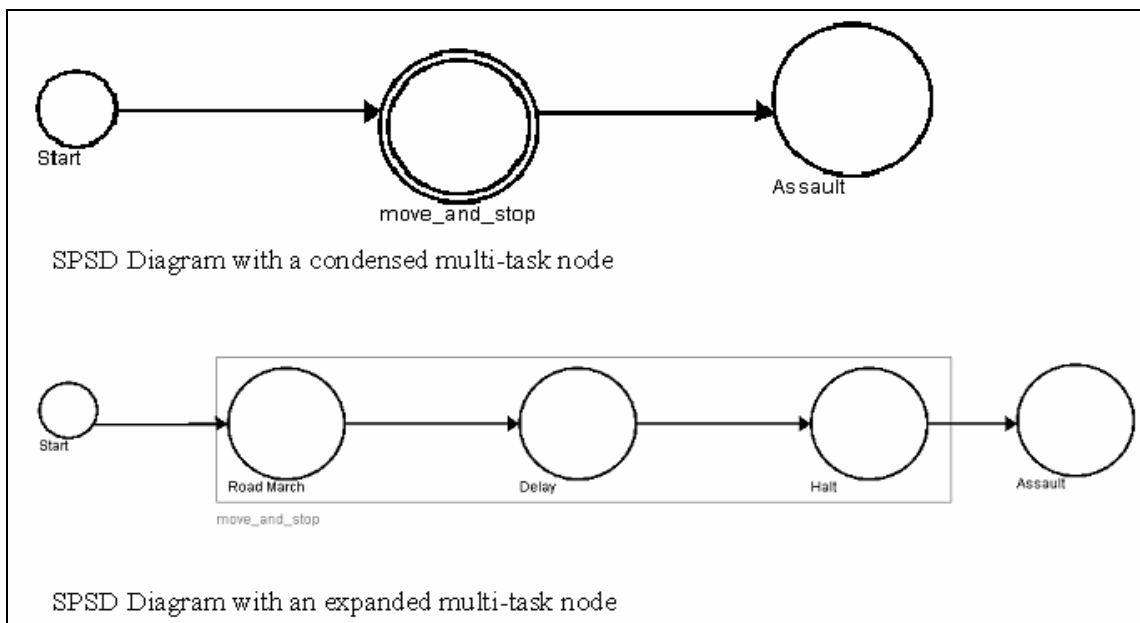


Figure 5. An SBSD diagram with a Multitask Node [2:82]

As a test bed to prove the effectiveness of the SBSD language, Bartley used scenarios from the One Semi-Automated Forces (OneSAF) simulation system, produced by STRICOM and used by the U.S. Army. Her results demonstrated that the language reliably described the scenarios with the desired requirements she discovered over the

course of her research. Unfortunately, she was only able to describe existing scenarios with the tool; time constraints kept her from performing the reverse: to create working OneSAF scenarios from the language alone.

Besides the SBS D language, Bartley created another visual tool used to show aggregation and composition of elements within a scenario as well as hierarchical order of the constituent entities. This other technique involves the use of treemaps. The treemap graph not only displays hierarchical information, but also provides a visual clue as to relative size and strength of entities involved in a scenario simulation execution run. Figure 7 shows a treemap of a OneSAF scenario.

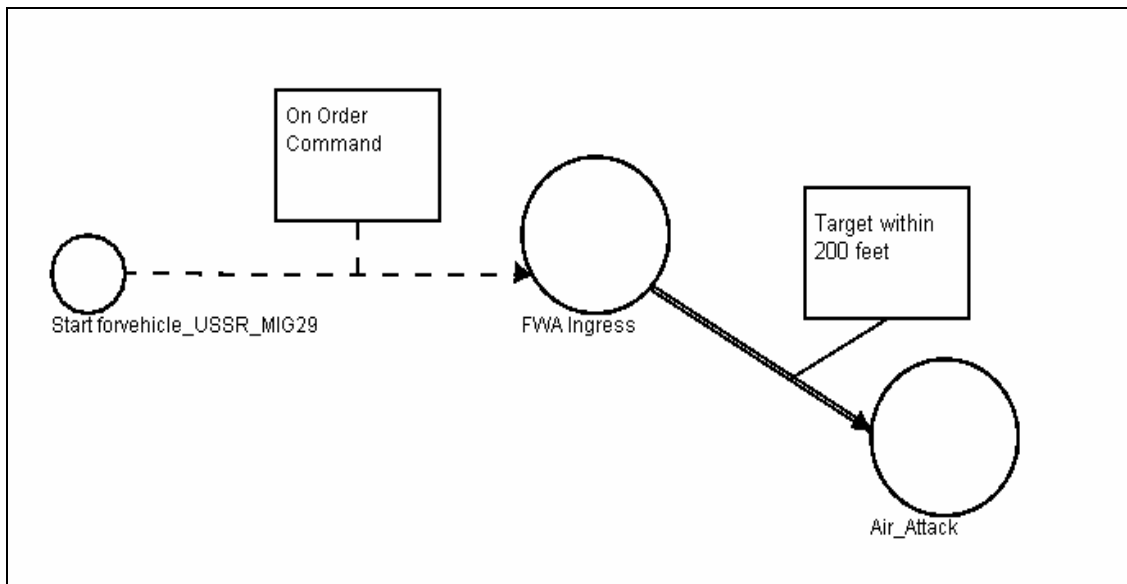


Figure 6. An SBS D diagram with a Temporary Reaction Transition [2:93]

Bartley's work is a breakthrough in describing the behavior of a scenario entity within a simulation, avoiding the pitfalls of other contemporary behavior depiction and pointing a way to solve some of the more difficult problems in rapid scenario creation.

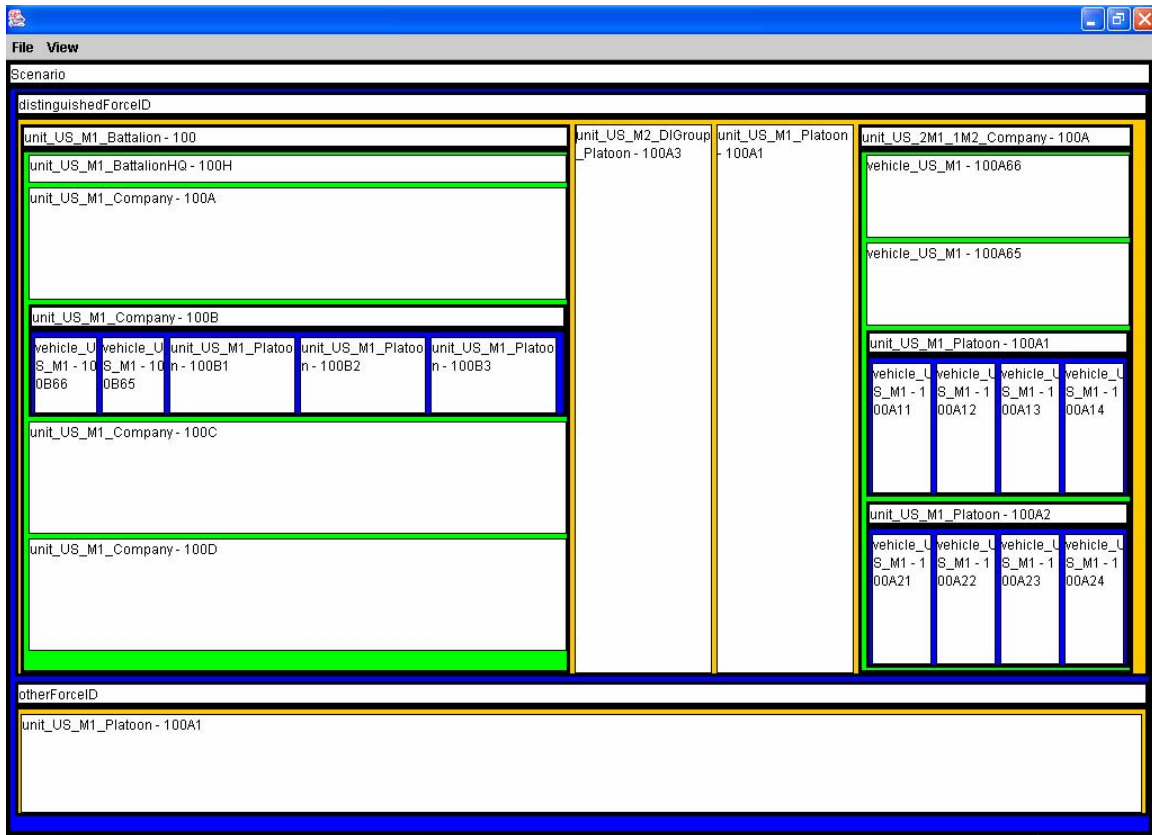


Figure 7. A Treemap of a OneSAF Scenario [2:114]

2.4 Other Research and Development

Many other organizations have added to the arena of simulation research. Some of these are not directly focused in scenario composition, but have potential application in that arena. An overview of a few significant efforts includes ModISE (emphasis on interoperability of simulation components), SimBionic (emphasis on graphical representation and description of entity behavior) and ADORA (a unified coherent modeling paradigm for object-oriented analysis and design). Each is discussed in turn.

2.4.1 Modular Interoperable Synthetic Environment (ModISE)

As part of efforts to develop a framework that encourages reuse of components, researchers at the Army Simulation Training and Instrumentation Command (STRICOM) and the Science Application International Corporation (SAIC) have created the Modular Interoperable Synthetic Environment (ModISE) system [3:1]. The goal is to develop an open architecture that allows users to create components that can be used as needed within new scenarios. The architecture supports components that use Defense Modeling and Simulations Office's (DMSO)'s high level architecture (HLA) data representation as well as others.

ModISE is based upon an interoperability engine (IE) that assembles the components required for a scenario from a repository that may be a web-based, distributed database or even a simple local database. The IE provides the virtual environment within which the simulation will execute. The actual execution of the simulation may be distributed depending upon the components used within the simulation run.

The repository contains Mapper/Translator/Modeler (MTM) components that represent entities, functions, environment—or a host of other objects. These MTMs must provide the interface for the IE to use them as well as metadata about the component so that the user can select the components required for the scenario [3:3]. MTM metadata can also contain metadata on what other components the selected component requires.

The user selects the required components—the MTMs—through a graphical user interface and builds an Execution Specification (ESpec). The ESpec is stored in the

repository with the MTMs. During simulation execution, the IE loads the ESPEC which provides the IE with the list of MTMs in the simulation along with the parameters for the scenario required for the run. The IE then loads the MTMs and then coordinates the simulation execution, including simulation timing, simulation event processing and interfacing data between components during execution of the simulation [3:5].

ModISE has been used successfully in trials with the Army and is offered as a product from SAIC. While it does promote reuse of components once built, it still requires a great deal of programmer effort to build MTM components. It does not deal with legacy components as of this research and there are still no automation tools to assist in the creation of MTM components. SAIC hopes to build tools to address these issues.

2.4.2 SimBionic

Daniel Fu and Ryan Houlette of Stottler Henke Associates discovered in the course of their research that developers typically rewrote the AI engines for games and simulations from scratch rather than leverage past development [7:81]. Developers they interviewed from various organizations claimed that the AI rules, which described behavior of computer-controlled entities, were radically different from application to application and therefore it was impossible to reuse or even to build an engine that could simplify the construction of these entity behaviors. Fu and Houlette hypothesized that this was a false assumption and began development in a simulation logic environment editor and runtime engine initially called BrainFrame[7:81], but later renamed SimBionic [16:7]. SimBionic's behavioral modeling is at heart an implementation of a hierarchical finite state machine.

SimBionic allows developers to create the behavioral logic of computer controlled entities using a graphical environment. The system is partitioned into development and runtime environments. The development environment (called the authoring tool in Figure 8) allows the user to describe the behavior of the entities in a graphical form with states and transitions (Figure 9). The states (shown as rectangles in the diagram) are tasks that the entity must perform—or states that the entity is in. The ovals are predicates that must be true for the entity to transition to the next task state. There may be multiple transitions from a given task state. They are numbered in order of test precedence; the predicate associated with transition 1 is checked first, next transition 2, etc. Task states may themselves be defined in another diagram; i.e., the task diagrams may be nested.

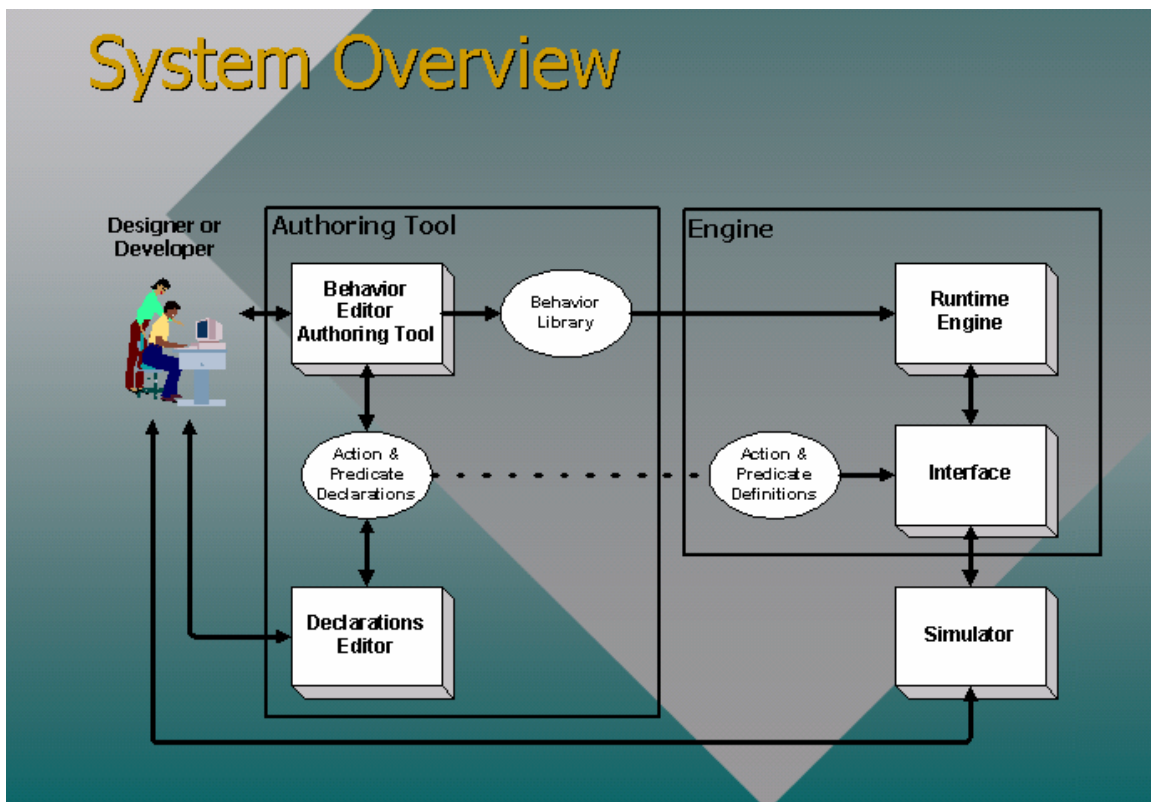


Figure 8. Overview of the SimBionic system [16:16]

The user creating the behavior diagrams doesn't need to be a skilled computer programmer; just an understanding of the entity's behavior is required. Thus, the behavior modeled by the entity can be described in a reasonably abstract manner, and is a positive step in assisting application domain experts a primary role in creating scenarios.

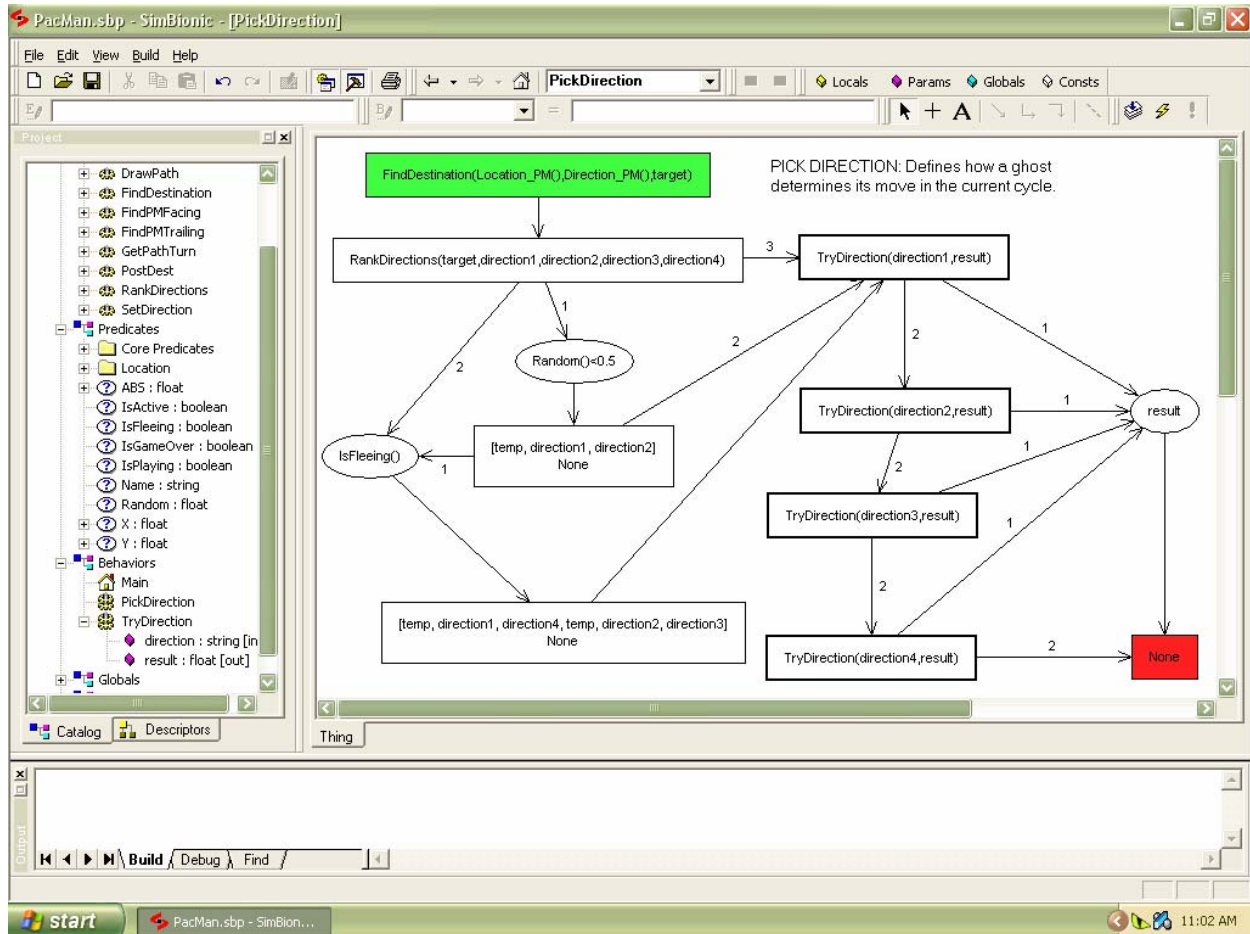


Figure 9. Example of SimBionic behavior graph [7:82]

Nevertheless, the authoring tool still requires a programmer to complete the connections of the behaviors for the simulation or game to operate with the SimBionic runtime module, shown in the “actions and predicate definitions” oval in Figure 8. So while domain experts can clearly play a more direct role in the design of computer-

controlled component behavior, computer programmers will still play a significant role in completing the simulation.

SimBionic does allow the creation of a behavior library where previously created entity behaviors can be stored for later reuse. However, the mechanism is very simple. The project's behaviors are stored and imported. The library is thus not an integrated behavior database. It is strongly dependent on users' awareness of existing behavior diagrams, what the semantic meaning of the diagram is and locating the behaviors diagrams for reuse. On the positive side, if a diagram is selected from the library, the system knows the dependencies as the diagrams must be a coherent unit. On the other hand, this can lead to multiple copies of behavior diagrams spanning the library, each of which may be inconsistent with other copies, much as using copies of computer source code across projects. A disciplined organization can create a separate database system (such as Oracle) that stores the semantic meaning of the diagrams along with the diagrams themselves; but this is an external activity requiring human interaction, not inherent in SimBionic itself.

SimBionic as of this research supports C++ and Java simulation programming. The bottom line is that while it is a significant tool to aid in the construction of new simulation systems, it provides little support to reuse scenarios in current systems.

2.4.3 Analysis and Description of Requirements and Architecture (ADORA)

The realm of software engineering, particularly object-oriented development, has been a fertile ground for developing visual tools to aid in the construction of large-scale systems. Currently, most practitioners have adopted the unified modeling language

(UML), defined by the Object Management Group (OMG), as the standard for analysis and design of these large object-oriented systems. UML is the culmination of many previous attempts to describe object-oriented systems including Object Modeling Technique (OMT). UML has many virtues: it depicts various models of a system; it is extensible; and it is an industry standard. Nevertheless, UML still suffers from certain difficulties from the perspective of software development as well as that of simulation scenario composition.

The single greatest flaw in UML is the lack of an integrated, cohesive overall view of the system from the separate modeling diagrams [9:426]. UML has class diagrams, use cases, collaboration charts and state charts (among others). These diagrams are loosely coupled, which provides a high degree of flexibility. Unfortunately, it also makes integration of these model aspects difficult. If a given design has inconsistencies that span across two different type of diagrams, the inconsistencies may be overlooked until later in development—where it is far more expensive in time and resources to repair.

Researchers at the Institute of Information Science, University of Zürich have been working on another modeling system that preserves the visual simplicity of UML while overcoming its inherent weaknesses. The ADORA modeling system is a major work in this area.

2.4.3.1 Description and Analysis of ADORA

The ADORA system is a highly integrated diagram that incorporates all of the aspects of the modeled system. It combines behavior, structure and collaboration all in

one diagram. In order to manage complexity, ADORA provides different views to the diagram. How ADORA is able to do all this is due to its unconventional concepts [9:426]:

1. ADORA works with *abstract objects*, not classes. An ADORA model places an object within a specific context of the system which typically constrains its behavior.

Commonalities of objects (which are grouped as classes in conventional diagramming methods such as UML) are modeled in *object types* in ADORA.

2. ADORA structures the modeled system in a strict hierarchical decomposition form. This permits a consistent and recursive approach to decomposing a complex system into simpler, more manageable subsystems. This is the key to ADORA's powerful cohesive approach to modeling systems.

3. ADORA uses an integrated model rather than a collection of models (such as UML). Thus the various aspects of a model (behavior, structure, collaboration, etc.) are all contained within the same model. However, not all aspects are visible at the same time as this level of detail would overwhelm the user. Instead, ADORA allows the user to select views of the model showing those aspect(s) in which the user is interested.

The ADORA system tool is composed of a base view which shows the abstract objects in relation to each other. This view can be very abstract or drilled down to the lowest level of detail described in the modeled system. The base view is shown in figure 10. Abstract objects are depicted as rectangles. In addition, ADORA models *abstract object sets* as well as objects. An abstract object set is a collection of one or more instances of the same type of abstract object, typically with cardinality to abstract objects in its context. In Figure 10, the RoomModule is an abstract object set.

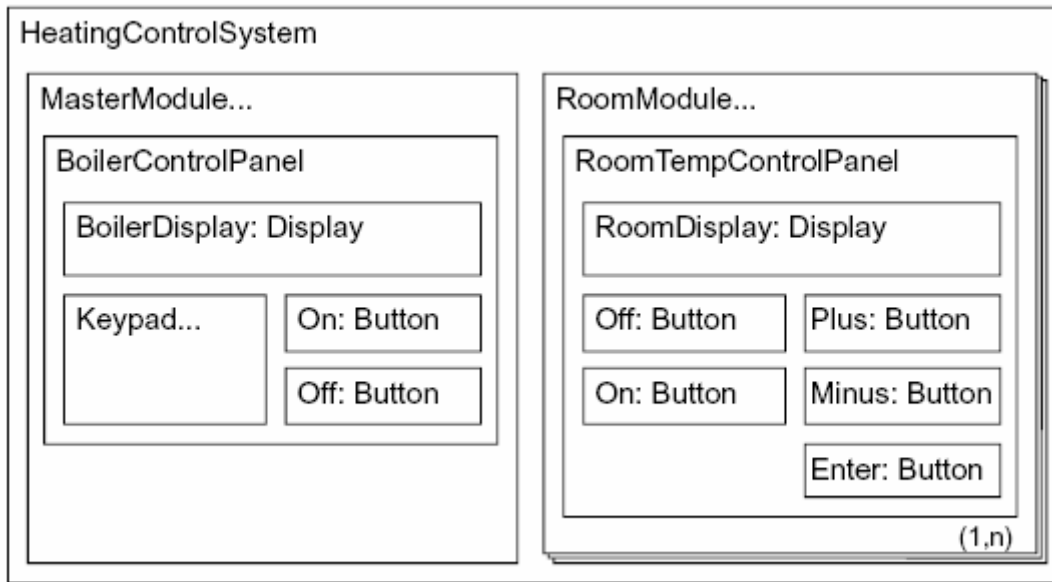


Figure 10. An ADORA base view [9:427]

A view can be partitioned such that the user views only a relevant subsystem. In Figure 11 the base view is the RoomModule partition augmented with a behavior view, shown both by state transition graphs and state transition tables. Note how the behavior is displayed in the context of the structure of the abstract object set.

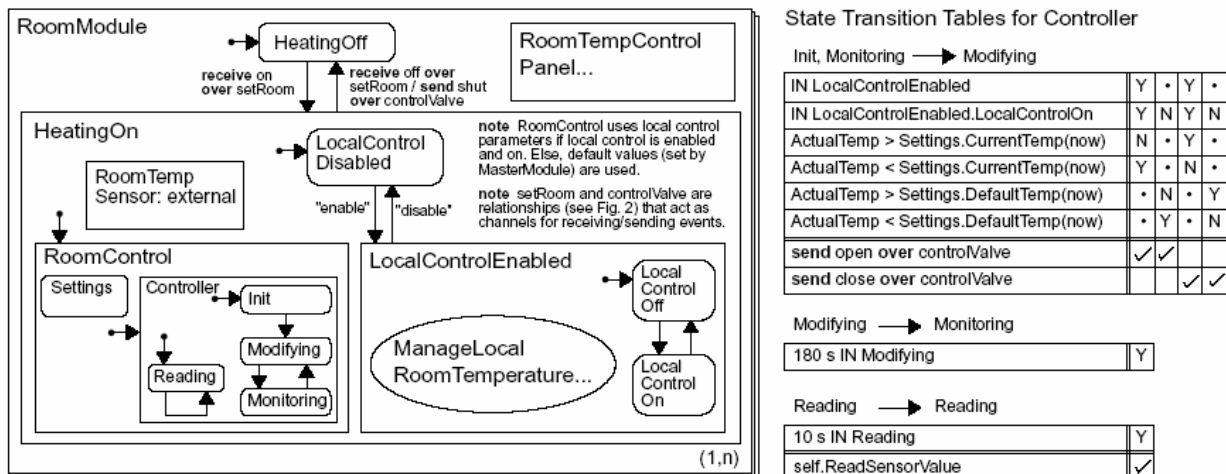


Figure 11. A partitioned ADORA base view (of the RoomModule abstract object set) combined with its behavior [9:432]

There are several other views offered in the modeling system. Nevertheless, the portions discussed show a potential for application within a scenario construction context rather than solely for analysis and design of object-oriented systems.

2.4.3.2 Conclusion on ADORA

The ADORA system has a great deal of potential as an analysis and design tool for software developers. Additionally, it appears to be an excellent starting point for further research in simulation-scenario development integration. The most significant weakness of ADORA for this usage is the fact that it is intended for software development, which tends to be deterministic; i.e., object associations are not probabilistic. Simulation components within a scenario, on the other hand, associate frequently by stochastic means; an object may or may not associate with another object.

A modified form of the ADORA system, using Bartley's Scenario-Based Specification Diagram language as an augmentation of the behavior description, is a promising candidate for a unified scenario composition language and tool. Such a tool would incorporate the hierarchical view of the components involved in a scenario (similar to Bartley's treemap view) in the structural base view and add the behavioral description (Bartley's SBSDD graphs) into one integrated model.

2.5 Summary

The wide use of simulation systems as an economical means of training personnel and developing effective tactics for a given scenario among many other practical uses has led to a very large number of simulation systems, each with its own distinct format of

representing entities. Most simulation systems have components that overlap those of other systems. Since a significant amount of time has already been expended to develop these entities and scenarios, it is obvious that component reuse would be highly beneficial. Unfortunately, the unique means to represent the scenarios and constituent components within each simulation system has been an impediment to their reuse.

Much research has been ongoing both at AFIT and other organizations to overcome this obstacle. AFIT researchers have worked on building a global format for converting scenario components from a simulation system specific format to another system format, allowing simulation scenario developers to avoid reinventing the wheel. AFIT research has developed rapid collation and presentation of these components through agent-based systems and AFIT has led the way in developing visual tools to map entity behavior within a scenario.

The effort now remains to apply the visual language with any necessary extensions in the creation of new scenarios, using existing scenario components, into a form for use of a live simulator system. That is the objective of this research.

III. Methodology

3.1 Introduction

As discussed in Chapter 2, there have been significant advances in the arena of composable simulation as well as visual depiction of components and behaviors. The goal of this present work is to bring together the two main threads of research conducted at AFIT: creating components with behaviors using visual language tools in a global or generic format that may be transformed into simulation-usable products. It also seeks to unify the visual aspects of the language into one coherent model. This chapter covers both the approach to accomplish these goals and the definition of the criteria to determine success.

3.2 Background

The following section discusses the two aspects of the problems—semantic meaning in existing scenarios and the process to create a unified visual language—to overcome in order to describe a solution plan.

3.2.1 Semantic Meaning in Existing Scenario Components

The information contained within a scenario component exists at two levels. One is at the face level—the information that the entity contains and accesses directly. The other level is semantic—meaning about the data, which is not typically contained within the entity or the data about the data; i.e., metadata. For a scenario developer to use an existing component in a new scenario it is not enough to know what is *in* a component; the developer must know *about* the component. For example, a component may have

behaviors (or tasks) stored within its definition. But the developer would be better served to know what *mission* the tasks are meant to accomplish—perusing the tasks would be arduous for the developer to find the proper component for reuse.

Thus, the visual language must provide a means to enter and display metadata about the scenario component, and—if and where possible—derive meaning, i.e., “semantics,” from existing scenario components. This semantic information must be stored in some form of a global scenario component database in a searchable format. Previous AFIT efforts—specifically Colonese [6], McDonald [12] and Breighner [4]—provided one approach to accomplish this goal.

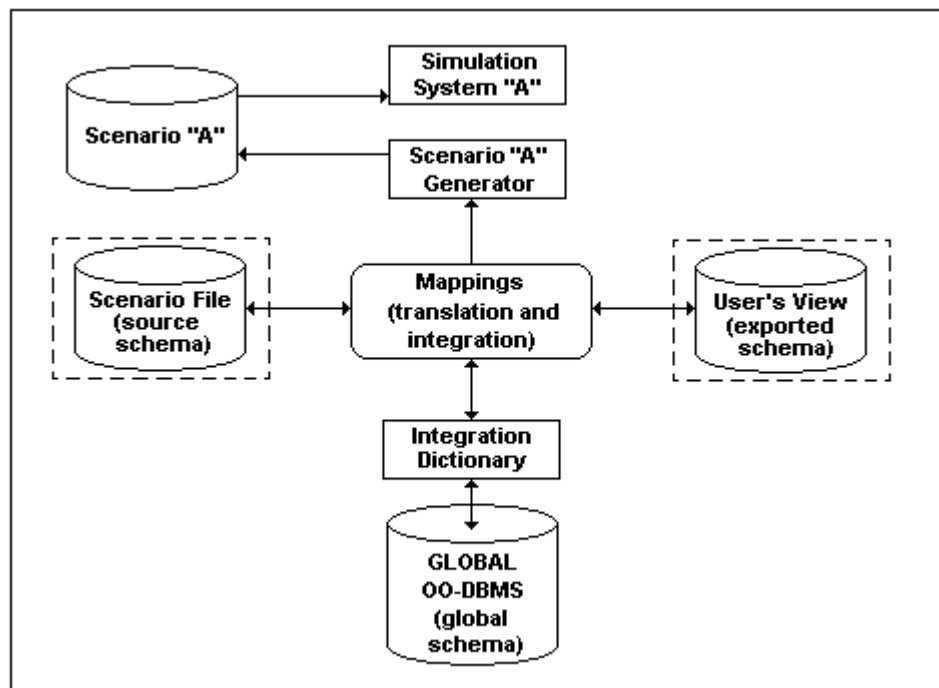


Figure 12. Generation of a Scenario [6:37]

The Colonese model for creating a scenario from the global object-oriented database for a specific simulation system is shown in Figure 12. This is the ideal form

she envisioned rather than the implemented form she created due to time and scope constraints. The Integration Dictionary in the figure provided the means of storing semantic data linked with the actual scenario component, or, in Colonese's terms, the "conceptual class" which contains the metadata as well as the "real class" [6:41,43].

Breighner's research [4:48-49] yielded an indexing scheme for rapid searching of the database. This technique would interface with the proposed global database to create a more useful end-user environment. Indeed, it is still a viable approach to store and retrieve scenario information.

However, generic data transfer between information systems has improved over the time since this research was accomplished. Therefore, the Colonese and Breighner methods are left to other developers to further refine the system for real world usage. This effort instead uses an XML approach for generic data storage, search and retrieval. This seemed a better approach due to the method of describing scenarios in the visual language rather than instances of scenarios.

By using this scenario depiction rather than scenario instance description, reuse is enhanced. Scenario developers can search for scenarios containing, for example, specific types of aircraft against certain types of radar rather than an exact instance of this specific unit's aircraft against this specific enemy radar located in this specific location. The user can select the generic form of the scenario and then use attributes of the components in creating a specific instance that is subsequently executed on a simulation system. The distinction is subtle, yet significant. The scenario provides entities (players) in context of the scenario, yet allows them to remain in a flexible state until the scenario developer

assigns those aspects to the scenario to fit the specific simulation execution. Tactics could, for example, be tested for effectiveness in different situations of terrain, enemy composition or weather, but still be the same basic scenario. The user-selected attributes make the difference.

3.2.2 Unified Visual Language

The unification process of the visual language requires adapting the resulting languages and tools of previous research efforts into one coherent model. Previous efforts have focused on specific subsets of scenario composition—structure of components, enforcing proper interfaces between components; or the behavior of components within a mission tasking. This effort instead utilizes a single point description of the generic players of a scenario with the behaviors that these generic players will perform.

Canli [5] initiated the visual language thread of research at AFIT with the focus on structure and proper interface between components. He found in his research, listed by Paige, Ostrof and Brooke, nine fundamental design principles of visual languages that are applicable to any visual language, regardless of its purpose [14:12-17]. These principles are simplicity, uniqueness, consistency, seamlessness, reversibility, scalability, supportability, reliability and space economy. Canli described these principles in part:

Simplicity: ... provides ease of learning, provides the ability to draw models by hand, and greater ease in creating software tools to support the language.

Uniqueness: If a language has the uniqueness property, it provides one good way to express every concept. This prevents ambiguities and redundant overlapping expressions in the models.

Consistency: ... of the language [i.e., the *language* must be consistent so that checks between views of a model may be reliably performed by mechanical means]. ... [C]onsistency between models [e.g., within UML] might become an important issue for a designer dealing with large-scale systems. It is questionable whether a consistency check for UML can be automated.

Seamlessness: This principle helps the ability to generate code from [the] model. It involves using the same abstractions in the model and in the textual language. This avoids a logical “impedance mismatch.”

Reversibility: The ability to generate a model from code contributes to the production of maintainable code and to the documentation.

Scalability: The language should provide mechanisms to handle large-scale problems. At the same time, these mechanisms should not detract from the design of small-scale models. ... [T]he language should provide concise mechanisms to define the fundamental abstractions, ways to hide details and grouping mechanisms.

Supportability: [The language] should be suitable for humans ... [,] implementable and supportable by software tools.

Reliability: ... To ensure reliability of the design, the language should provide support for automatic consistency checks via the grammatical rules of the language.

Space Economy: The models should take as little space on screen or page as possible to reduce distractions caused by search and browsing. [5:64-65]

Previous AFIT research on the visual language thread focused on these principles with varying levels of emphasis. As the primary goal of this research is a unified visual scenario composition language potentially capable of creating scenarios usable by any simulation system with a global format transformation module, the main emphases are on simplicity, consistency and scalability without sacrificing the achievement of the other principles by included components from the body of prior work

Bartley’s research focused on behavioral descriptions using nodes (representing tasks or activities) and transitions between these tasks. In her discussion of required properties of a visual language, she described several essential properties for a viable visual language [2:70]. These include: (1) “the specification of attributes in the representation of activities ... [so that] the user [is] able to specify certain attributes of the behaviors assigned to entities;” (2) ability of the user to place constraints on the transitions between tasks; (3) composition of new assignable activities from existing constituent activities to facilitate ease of use in assigning the same task to multiple

entities. These properties, which she listed and implemented in her behavior description language (SBSD), are incorporated into the visual language tool of this research as well.

Finally, while ADORA was not developed at AFIT, it provides a conceptual framework by which simulation entities, structurally and behaviorally, can be designed with a strong enforcement of context within the scenario. Thus, the language and tool environment produced in this research incorporate the earlier languages and unify them into one coherent integrated model with an ADORA-based system as the consolidating component of the proposed visual language system. Indeed, the ADORA approach makes the generic scenario description possible. Note that ADORA is not implemented itself in this research nor is the ADORA toolkit utilized in the environment; instead the underlying concept and approach used by that system are used as a basis for this new system.

3.3 Design Principles

Since entities and behaviors/tasks can be modeled naturally in an object-oriented manner, an object-oriented approach is the logical one. Scenario entities are objects; often composite or aggregate in essence; e.g., an aircraft with its constituent radar and weapons systems. Tasks to be performed by the entities within the scenario can also be modeled as objects. The object system would be the individual tasks to be performed by the entity in a sequence, with tasks potentially organized into conglomerate objects for ease of understanding or reuse by the scenario developer. Transitions may be modeled as objects containing conditions that must be satisfied in order to properly sequence the next task object.

Additionally, entities and behaviors in the scenario are heavily dependent upon context—the associations and roles of components with respect to each other within a mission case. Thus, the visual language was based on a modification of an object-oriented model that emphasizes the objects in their context rather than primarily their commonality of structure. This requirement was fulfilled by adapting ADORA methodologies and concepts to the language and tool.

The adoption of an ADORA-like approach to entity modeling provides another important aspect of the unified visual modeling language: coherence. The design language and tool supports a clear unity of views. This is in contrast to the approach by UML, whose flexibility of disjoint diagrammatic views allows too much freedom of expression and can permit ambiguity or even contradiction that is not easily detected [5:51]. The unified visual language thus fulfills the principles of uniqueness and consistency, while retaining fidelity to the other language principles described in the previous section.

Finally, the unified visual language leverages the accumulated knowledge of previous research. Wherever possible, components of the language and the data store are adapted from those prior works. This current research is thus of an evolutionary rather than revolutionary nature.

The main design principles are, therefore, an object-oriented approach, focusing on a coherent model with supporting views, adopting results of earlier research into a new synergistic system.

3.4 Evaluation Criteria

Bartley developed as a set of evaluation criteria the following, based on the criteria developed by Canli and other sources:

Expressiveness: Were certain aspects or properties impossible to express? If so, what? Were some things difficult to express?

Frequency of errors: What are the most common errors and the frequencies of those errors? Why did those errors occur? How can they be avoided? Where is the potential for errors?

Redundancy: Was redundancy present in the models? Is it possible to identify different types of redundancy? How can redundancies be avoided? Where did the redundancy occur?

Locality of change: Do changes propagate through the models? If so, what are the causes, and can they be avoided?

Reusability: Do the models enable reusability?

Reliability: Do the models enable consistency checks? If not, why and how can the inconsistencies be avoided?

Translatability: Are the models consistent and expressive enough to be used as an input to a simulation tool?

Compatibility: What is the distribution of results of the above criteria? [2:75]

These criteria are equally pertinent to this research as well. Most of these criteria are binary—yes/no—rather than quantitative in nature. Additionally, the test environment for this research was the OneSAF simulation tool using case studies. An emphasis was placed particularly on the reusability aspect of the language and environment tool, although reliability (through grammar descriptions) and translatability are also strongly emphasized. It is to be noted that the translatability is not proved through creation of input for an actual simulation system, but rather through a demonstration that the resultant output contains the necessary components and their relationships to each other in a format as to permit a mapping to a simulation-specific scenario file.

3.5 Summary

Many of the foundational concepts were laid down ably by previous researchers at AFIT. This work attempts to extend their efforts into a unified whole; specifically, the visual language and tool attempts to fuse these components into one consistent mechanism by which the different aspects (structure and behavior) can be used to develop generic scenarios which are stored in a searchable format. These generic scenarios would then be processed by a transformation engine to produce artifacts that are executable on the intended target simulation software system.

IV. Conceptual Design of the Visual Language

4.1 Introduction

The visual scenario language is implemented in three aspects that correspond to the research software tool's three sections: abstract entity structure, entity behavior and concrete entity binding. The abstract entity structure section describes the scenario in terms of *players within a context of the other players in the scenario*. They are abstract in that they are not bound to an actual instance of an entity, allowing the scenario to be described generically. The binding to actual simulation entities for an execution run is performed in the concrete entity section. The mission tasking associated with each entity in the scenario is described in the entity behavior section.

The language in this document is described primarily in terms of the implementation tool, but it is important to keep in mind that the language is meant to aid developers in composing scenarios in a generic sense. The tool provides a convenient method of composing scenarios as well as a method of validating the concepts of a scenario. A complete tool—one designed for actual field use rather than strictly research—would additionally have a translator that transforms the scenario into scripts usable by live simulation systems to execute simulation runs. Nonetheless, scenario developers could benefit by utilizing the language to design their scenarios, because it enforces constraints which naturally are associated with interacting objects.

4.2 Abstract Entity Structure

Abstract entities are the key to the generic nature of scenario composition in this language and tool. An *abstract entity* represents a component that is a participant or player in a context within a scenario, but without the details that lock the entity—and by extension, the scenario—to a specific simulation instance. An abstract entity is represented in the language as a rectangle containing its identifier and its type, separated with a colon. An abstract entity symbol also includes a circle in the upper left corner to indicate the presence (or absence) of an associated behavior map. Figure 13 depicts an abstract entity and an abstract entity set (defined later in this chapter). An example of an abstract entity would be an aircraft assigned a bomb attack role in a generic scenario mission. (To clarify misunderstanding of common terms, this document will use *entity* to refer to an abstract entity and either *concrete entity* or *actual entity* to refer to an entity with simulation-specific details required to create simulation execution scripts.)

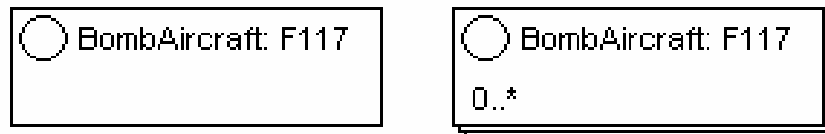


Figure 13. Abstract entity (left) and abstract entity set (right)

In comparing scenario composition to a software development effort, an abstract entity is conceptually closer to an abstract object than a class. A class describes the structure of objects in an object-oriented system; it does not describe roles of different objects of the same class within a system. In a similar way, scenario developers are not

interested solely in the structure of a specific class of aircraft (for example), but rather in the role that aircraft of a type are playing within that scenario. Figure 14 demonstrates the difference between an abstract object-oriented approach and a class-oriented approach to viewing a software subsystem as modeled with ADORA. Note that while the visual scenario language doesn't use ADORA itself, it does use similar techniques to emphasize the context of the entity within the scenario. But the example clearly demonstrates the differences between the two approaches.

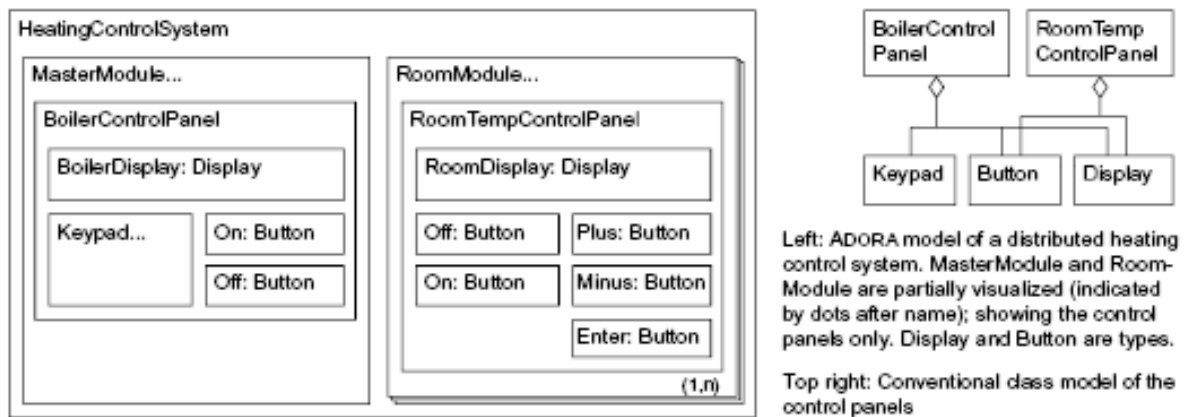


Figure 14. Abstract object-oriented view versus class-oriented view [9:427]

While the abstract entity approach aids in the understanding of roles played by the abstract entities within a system, the structure of those same entities is still extremely important. The next section covers the description of abstract entity structure.

4.2.1 Abstract Entity Attributes and Entity Types

The entity in the visual language is abstract in that it does not have attributes in the sense that a concrete entity has attributes—values that determine the actual entity's state at a given point in the simulation run. This does not mean that the abstract entity

lacks attributes altogether. It may have attributes that contain appropriate parameter types and constraints that are germane to a concrete entity performing the specific role in the scenario. There typically are other attributes shared in common with all other entities of that type regardless of contextual role. Such attributes, also in the form of constraints, are depicted in the form of an *entity type*, which are covered below. In any case, the abstract entity's attributes are not set to specific values. They are described as value ranges, enumerations or other similar forms. Of course, it is conceivable that a particular attribute may only have one value; in this case the constraint is an enumeration list containing only one item. All entities must have an entity type, even if the entity type contains no attributes. Proper entities (as opposed to reaction entities, defined in Section 4.2.2) must also have an identifier.

The *entity type* is analogous to a class in an object-oriented system. Entity types also have attributes, as mentioned above. As these are abstract entity types, the attributes are also in terms of constraints and not an actual state of a concrete entity of the specified type. Just as classes in an object-oriented software system have mechanisms to support inheritance, entity subtypes may be defined from parent types. A subtype inherits all of the attributes/constraints of the parent type. Unlike the class, however, a type only lists those attributes and constraints that are common to all entities of that type; there are no methods in the entity type, as these are entities within a simulation scenario and not software development artifacts. Entity types are not modeled visually using abstract objects; they are more naturally represented by a modification of the UML class notation; i.e., with the limitations mentioned.

As indicated earlier, those attributes that are closely bound to the role of an entity rather than the entity as a whole are defined in the abstract entity itself. However, an attribute that is common to all entities of the type may require more narrowly defined constraints for a given entity by the role the entity is in; in this case the attribute would be duplicated in the entity's attributes with the narrower constraints. This is roughly analogous to polymorphism.

It is common within scenarios for entities to be described in a collective or aggregate form as well as singleton entities. An aggregate would consist of entities of the same type and the same mission, allowing the simulation system to deal with the aggregate as a single item rather than as several separate entities. As the simulator would “move” the aggregate as one, the visual language should allow a representation of this aggregate entity of the same type as the individual entity to simplify assigning constraints and mission behavior tasks. This is discussed next.

4.2.2 Abstract Entity and Abstract Entity Set

Abstract entities are divided into two varieties: the *abstract entity* proper and the *abstract entity set*. The sole difference between the two is that the abstract entity set has a user-defined cardinality. Figure 13 shows the graphical representation of the abstract entity set juxtaposed with the abstract entity. The cardinality is in the lower left corner of the entity set. The “multi-page” appearance is another visual cue to the user of the entity set. The default cardinality of an abstract entity set is “0..*” which is interpreted in the same manner as UML: minimum zero, maximum some arbitrary value; i.e., “zero or

more” entities. An abstract entity set with cardinality “1..1” is exactly equivalent to a proper abstract entity.

Allowing this alternate form—the equivalence of an entity set with cardinality of 1..1 and an abstract entity of the same type, behavior and constraints—technically violates the principle of uniqueness by having a redundant form of expressing a single entity. Nevertheless, it is a reasonable compromise in that it allows the scenario developer flexibility to change the cardinality of the entity set while the scenario is under development.

Because of the close relation between the two types of abstract entities, the visual language tool provides a means of converting entities from one form to the other. Converting an abstract entity to an abstract entity set changes not only the appearance of the modeled entity, but adds the cardinality attribute. By default, cardinality is 0..* which may be changed just as the case of normally created abstract entity sets. Converting an entity set to an entity causes the cardinality to be lost as a distinct attribute; naturally, the inherent nature of the entity is that it is of cardinality 1..1. But the developer must have this capability in order to select the appropriate entity form once it is determined which best fulfills the scenario design.

A significant benefit provided by the conversion capability is the added flexibility to reuse scenarios. If an existing scenario needs to produce another simulation run but with a different number of players within a given role, the entity-entity set conversion allows such an adaptation to be easily made. This is possible by keeping in mind that all abstract entities represented by an abstract entity set share the same attribute constraints

and mission tasks. In other words, the association of entities represented by an entity set is that of role as well as type.

There is an additional case of entity: the *reaction entity*. This is an entity that does not have a role itself within the scenario. It is strictly for the purpose of creating reaction transitions, described in Section 4.2.4. A reaction entity is distinguished from an entity by having a blank identifier. A reaction entity must have an entity type. A reaction entity may not have a behavior assigned to it nor be part of nested components.

4.2.3 Nested Abstract Entities

Abstract entities and entity sets (but not reaction entities) may be nested within each other. This corresponds to aggregation. Nested entities are components of the entity in which they are contained. For example, an aircraft entity may contain a radar system, a complement of air-to-air missiles, a complement of air-to-surface missiles and a Vulcan rotary machine gun (see Figure 15). The containing entity has by default a communication link to its nested components, so that commands may be sent from the aggregate entity to the subcomponent entity (e.g., command to air-to-air missile to fire) and information flows from the subcomponent to the aggregate entity (e.g., radar detects aircraft at bearing X, range Y). Communication links are covered in greater detail in the next section.

As entities or entity sets in the scenario grow in complexity due to nested aggregation, it may become desirable to remove from view some of the details of the system. The visual language tool provides a means to perform this essential task of *information hiding*. Figure 15 shows on the left hand side the same entity but in “hide

details” mode. The entity with details hidden informs the user of this situation by displaying an ellipsis after the entity identifier (i.e., entity name: entity type). The visual language tool additionally shades the entity gray to call attention to the “hide details” state, but it is not a requirement of the language proper to use color.

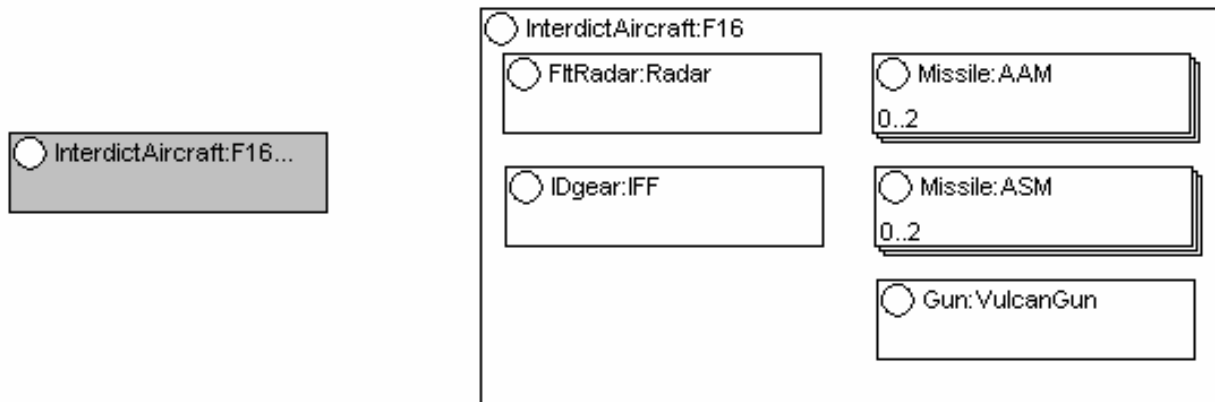


Figure 15. Nested abstract entities

The tool provides two ways to accomplish the hide/show details functionality: one is to double click the abstract entity or entity set that the user desires to change the display details state; the other is to click the hide/show button on the entity panel toolbar with the entity in the selected state. If an entity doesn’t contain nested components, the hide/show details command is ignored. Details at any level within nested components may be hidden or shown at the respective level. If, for example, an entity (referred for illustration’s sake as the base-level entity) contains an entity (referred to as level-one entity) which itself contains an entity set (referred to as level-two entity set). The user may hide the details of the level-one entity; i.e., hide level-two entity set. Later the user may hide the details of the base-level entity. If the user later shows the details of the base-level entity, level-one entity is still in a “hide details” state.

The ability to indicate hidden details is also useful for developers drawing entities by hand using the language notation: if details are known to exist, but are either uninteresting at this point or are as yet undefined, the developer can draw the ellipsis to indicate the presence of hidden details of the entity.

4.2.4 Transitions and Abstract Entities

The abstract entity diagram allows two types of transitions: communication transitions and reaction transitions. The former is based on an extension of the nested entity concept. An entity has access to its nested entity components. The *communication transition* provides a way for entities that are not nested with respect to each other to send and receive messages. Reaction entities may not have communication links.

Figure 16 shows the example of an aircraft containing missiles that use the aircraft's radar system for targeting. The radar is not part of the missile system, but the missile must be able to receive targeting messages from the radar.

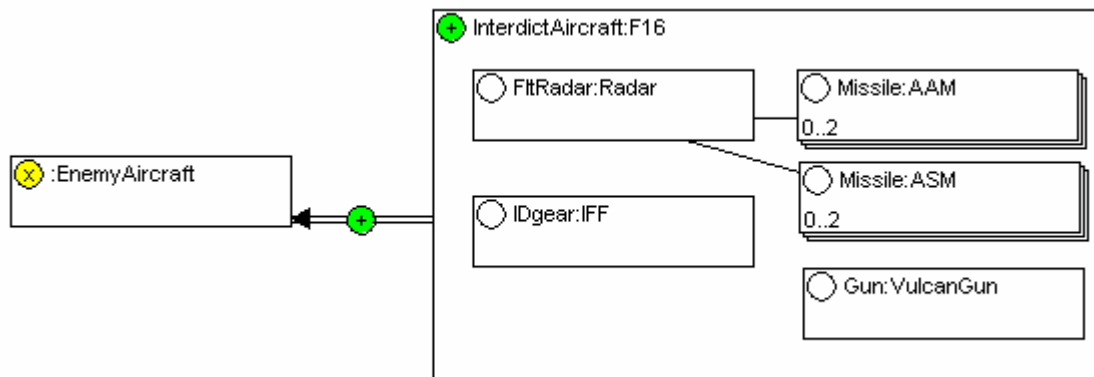


Figure 16. Communication and Reaction Transitions between Entities

If a conditional transition of any type (described in greater detail in Section 4.3.2) refers to a communication message as part of the conditional test, then for the test to be

checked there must be a communication line between the entity and the source of the message; otherwise the condition will never be triggered. As referred earlier, a nested entity inherently has a communication line with its enclosing entity; however, two entities that are both contained within the same enclosing entity (i.e., sibling entities) do not automatically have a link.

The *reaction transition* indicates that the entity represented by the source of the transition will react at the detected presence of the entity represented by the destination of the transition. Typically this is a reaction entity, which means that the entity will react as described to the detected presence of any entity of the type represented by the reaction entity. If the destination is an abstract entity of entity set (i.e., not a reaction entity), then the entity will only react to an entity in the role specified by the abstract entity; this is an unusual case.

The reaction behaviors may be defined in the behavior map associated with the reaction transition, indicated by the circle in the center of the reaction transition. These behaviors may in turn be referenced in the mission behavior map of the entity. Reaction behaviors defined in this map are not automatically executed; they must be selected from within the entity's mission behavior map. This allows the scenario developer to consider these contingencies in their context, much as a planner may create a contingency checklist for use in an actual event.

4.2.5 Abstract Entities and Behavior Maps

While the reaction transition of the abstract entity diagram may have associated behaviors defined within it, the abstract entity itself logically has an associated behavior

or mission tasking. The behavior diagram language used in both cases is described in the next section.

The behavior of an abstract entity or entity set may be an empty or undefined description. When an entity is created, it does not have an associated mission tasking defined for it until one is created for it. The entity may be left with the behavior map in an empty state. If a run-time simulation scenario is generated for such an entity, the behavior is interpreted as “the entity will remain idle” throughout the execution of the scenario in the simulation run.

The visual language tool checks for syntax errors in the behavior map and provides an “at-a-glance” indicator as to the status of a given entity’s behavior tasks: the circle in the upper right corner of the entity or entity set (see Figure 17). If the behavior is undefined, the circle is blank. If syntax errors exist, the circle is colored red with an exclamation point in the center. If the behavior is defined and the system detects no syntax errors, the circle is colored green with a plus sign in the center. If an entity is strictly an entity type (i.e., it has no role as a distinct entity in the scenario), then the circle is colored yellow with an “x” in the center. Syntax errors are covered in the entity behavior language section.

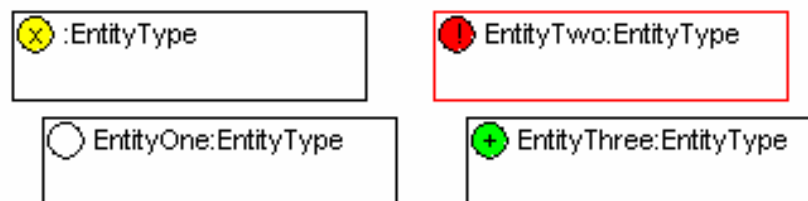


Figure 17. Behavior Map Status of Abstract Entities

4.3 Entity Behavior

Entity behaviors are defined using a slightly modified form of Bartley’s Scenario-Based Specification Diagram (SBSD) language. A brief description of this behavior language is provided in Section 2.3.2. This section provides a more detailed view on SBSD as it is implemented in this visual scenario language. The complete specification of the language as developed by Bartley, including an analysis of its effectiveness with other behavior mapping techniques, is defined in [2:77-97].

4.3.1 Nodes

The nodes in the SBSD language correspond to tasks performed by the entities within the scenario. The two user created types of nodes are atomic nodes and multitask nodes. Atomic nodes represent the basic task element performed by the entity—the task cannot be divided into lower levels of subtasks; hence, the term *atomic node*. As Bartley emphasized in her work, this doesn’t mean that an atomic node is necessarily a simple tasking; it may be quite complex and involve many subtask threads. Nevertheless, the atomic node is the lowest level task that a scenario developer may refer to in creating a scenario [2:80].

Additionally, there exists one more type of node, the *start node*. These nodes are not created by the developer and are strictly to mark the first task in a behavior sequence; the sequence may be a multitask node or the assigned behavior map.

All nodes, except the start nodes, must have at least one transition into them. The visual language tool considers the existence of unreachable nodes within a behavior sequence to be a syntax error.

4.3.1.1 Multitask Nodes

Multitask nodes are sequentially processed tasks contained in one task unit. Multitask nodes are intended to increase abstraction by hiding unnecessary detail from scenario developers once they are defined. Multitask nodes also provides a means of reuse: once common tasks are placed in an execution order, they may be referred to by the multitask name in other entity behavior maps [2:82]. Figure 5 in Section 2.3.2 provides an example of a multitask node in both condensed and expanded states.

A multitask node has only one exiting transition. This may be either a regular transition or it may be implied by a temporary reaction transition; i.e., upon completion of the multitask sequence, the task that was the original source upon execution of the reaction is resumed. Multitask nodes may contain regular or conditional transitions between task nodes within the multitask node. If alternate task sequence branches exist within a multitask node due to conditional transitions, then the exit transition path of the multitask node is traversed upon completion of any taken branch within the multitask node; i.e., there is an implied “end” task that all branches within the node ultimately transition to after the branch tasks have been performed in sequence. A more detailed discussion of transitions is given below.

The multitask node is therefore a container to conveniently hold the subsequence of tasks and not an executable task in itself. It provides another method by which the language promotes scalability via information hiding. Once the developer has defined the sequence of tasks along with the corresponding transitions within the multitask node, the details of the sequence can be hidden by “condensing” the multitask node and

referring to it in the condensed form. As in the case of hiding details of complex abstract entities, the tool provides two ways to accomplish the condensation/expansion: one is to double click the multitask node; the other is to click the hide/unhide button on the behavior panel toolbar with the multitask node in the selected state.

A multitask node must contain at least one task node in addition to the start node. The visual language tool considers the case of a multitask node containing no task nodes to be a syntax error. There is an exception to this: the reaction pseudo-task described in Section 4.3.2.2. This is permitted because the task is not really undefined; the definition is associated with the abstract entity reaction transition.

The visual scenario language modifies the original description of Bartley's multitask node in adding an explicit *start node*. This modification provides a consistency of the mapped behavior within the multitask node with respect to the top level behavior map. In fact, a complete mission behavior for an entity may be reused as a multitask node in the mission of another compatible entity, provided that there are no reaction transitions in the original behavior map. Allowing reaction transitions within multitask nodes were considered, but rejected due to the requirement of permanent reactions not resuming the original task upon completion of the reaction task sequence. This would violate the requirement of all branches in a multitask node ultimately traversing the exit transition path; i.e., resume the original task. The temporary reaction would not violate this rule; however, then the language would violate the principles of simplicity and consistency. The scenario developer would have to remember that temporary and permanent reactions are allowed—or not—in similar cases. Since the notation and the

concept between the two reaction types are so strongly linked, it was judged better to treat the two reactions in the same way with respect to reuse of scenario behaviors as multitasks in other entities' behavior maps, to conform to the principles of consistency and simplicity.

4.3.1.2 Nodes and Attributes

As indicated earlier, atomic nodes may also have attributes. These attributes are the parameters that the modeled entity requires upon entry to the task in order to perform the task within the context of the mission. The parameters should fall within any specified constraints as applicable.

Nodes in the visual language tool vary slightly from Bartley's notation in that a count of the number of attributes associated with the atomic node is displayed in the center of the node, as shown in Figure 18 (nodes Air launch, Approach and Assault). This provides the developer a visual cue that the atomic node has attributes associated with it—parameters that require specific values upon entry to the task during a live simulation run from the scenario.

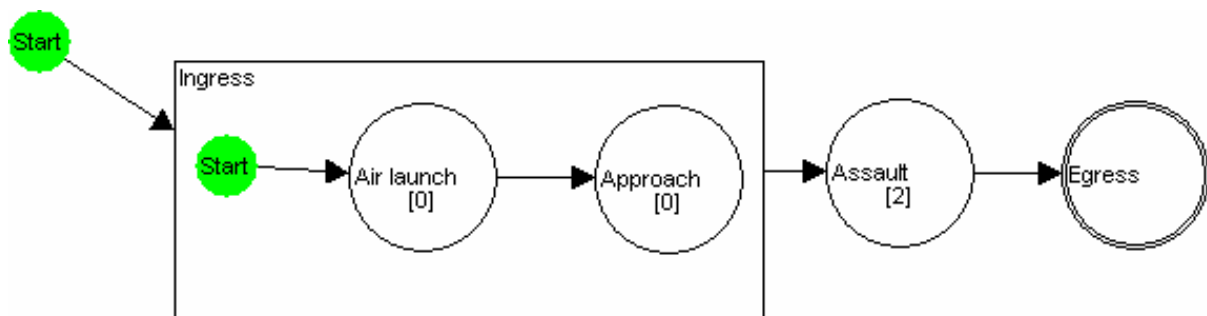


Figure 18. Atomic and Multitask Nodes

Multitask nodes do not have attributes due to their nature as “macro-nodes.” Multitask nodes do not actually describe a discrete task event, but rather group these events into a sequence. If a task contained within the multitask node requires parameters, that atomic task within the multitask provides the proper location to define the attribute parameters; in Figure 18, Ingress and Egress are multitask nodes, in expanded and condensed states, respectively. Any or all atomic nodes within a multitask node may have parameters, at the discretion of the scenario developer.

4.3.2 Transitions

Transitions are used in the Scenario-Based Specification Diagram to indicate the next task an entity is to perform within its assigned mission in the scenario. There are four types of transitions: regular and conditional transitions, and permanent and temporary reactions. A *regular transition* is the base form and indicates which task is to be performed in the sequence upon completion of the current task. As a consequence, a node may have at most one regular transition leaving the node, although there may be several conditional transitions leaving the node. (There is an exception to this when temporary reactions are involved; the exception is described in the reaction transition section below.) Furthermore, cycles are not permitted due to the task frame nature of the behaviors. Thus, a task node may have only one transition entering it.

4.3.2.1 Conditional Transitions

While a given node may be the source of at most one regular transition, it may have several conditional transitions leaving it. Each conditional transition has an

associated condition, a mandatory element, which must evaluate to true for the transition to be traversed; i.e., the condition is essentially a guard condition. The condition may be simple or compound; i.e., AND/OR operators chaining simple condition clauses. While Bartley left the issue of primacy of conditional transitions undefined (due to the nature of OneSAF, which does not provide a means of identifying primacy), this language modifies the definition to provide the developer the means to create the order that the conditions are to be evaluated, provided that the simulation system allows this capability. The first condition that is satisfied in the evaluation order has its transition traversed. Figure 19 shows a few conditional transitions leaving a node (Task One). The conditional transition may be identified by name (if provided one by the developer; e.g., transitions ConditionTwo and ConditionFour) or by condition if not (e.g., condition transition to node Task Three), preceded by the evaluation order.

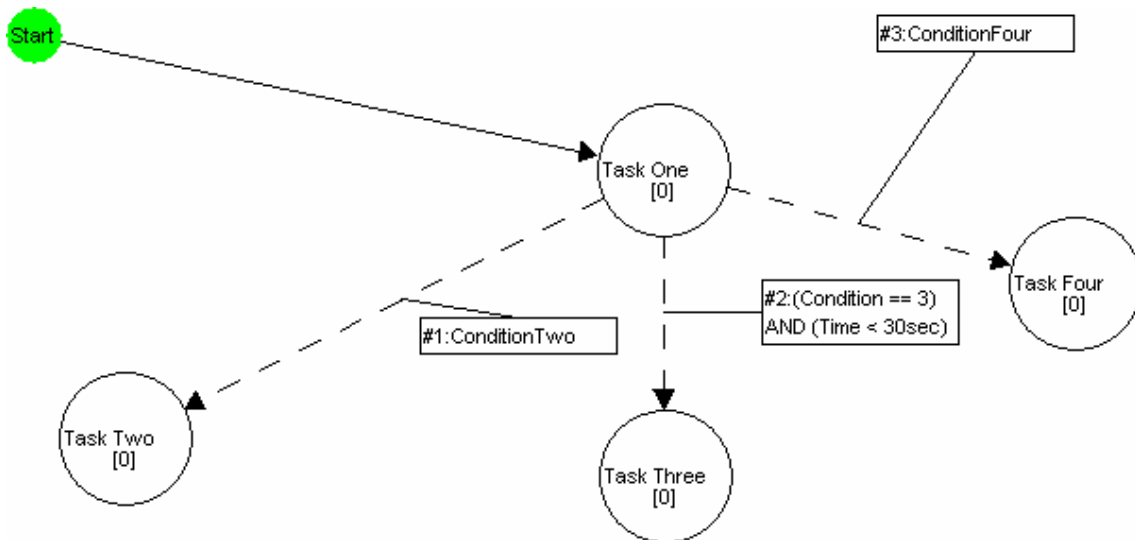


Figure 19. Conditional Transition between two Atomic Nodes

The conditional transition has two subtypes besides the basic form: the two reaction transitions. These are covered next.

4.3.2.2. Reaction Transitions

Reaction transitions are a variation of conditional transitions. They respond to external events that are not part of the original mission plan and are analogous to contingency tasks. There are two types of reaction transitions, permanent and temporary reactions. Figure 20 provides an example of the reaction transitions (EventOne is a permanent reaction, EventTwo a temporary reaction).

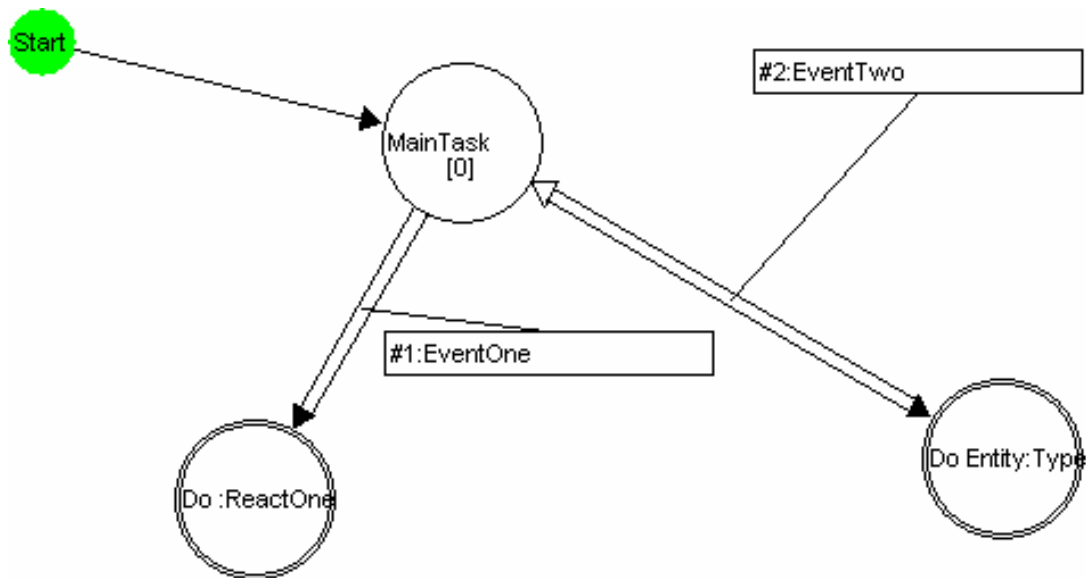


Figure 20. Permanent and Temporary Reactions

Permanent reactions interrupt the normal processing of the task sequence and due to the nature of the contingency, the original task sequence is abandoned and the contingency task sequence executed. As indicated in Section 4.3.1.1, this requirement means that it may only be created in the top-level of the behavior map; i.e., it may not be

created inside multitask nodes, as all task sequence branches within a multitask node ultimately resume with the task succeeding the multitask node.

Temporary reactions also interrupt the normal processing of the task sequence, but upon completion of the contingency tasks, the original interrupted task is resumed. As a consequence, the temporary reaction performs the function of the regular transition with respect to the reaction task; i.e., when the reaction task is completed, the interrupted task is the next task in the sequence to be executed. Thus, the destination node of a temporary reaction may not have a regular transition originating from it. A variation from Bartley's notation of the temporary reaction emphasizes this by indicating the source (interrupted) node with a "hollow" arrowhead rather than identical solid-fill arrowheads on both ends of the transition. Like permanent reactions, temporary reactions may not be created inside multitask nodes; this is strictly to comply with the principles of simplicity and consistency.

The destination node of either type of reaction may be a pseudo task: the destination refers to a sequence defined in the behavior map of the reaction transition of the abstract entity diagram as alluded to in Section 4.2.4. Such reaction destination nodes are defined by the label "DO [reaction entity identifier:type]," where the identifier and type are of the destination entity of the reaction transition in the abstract entity diagram; e.g., the destination multitask node of reaction EventTwo in Figure 20. The identifier may be omitted if the destination of the transition in the abstract entity diagram is a reaction entity and therefore has no identifier; e.g., the destination multitask node of reaction EventOne. Additionally, the reaction transition itself, like all other conditional

transitions, requires a triggering condition. If the reaction is based on the presence of the reaction entity, then the condition may be set to “REACT [reaction entity identifier:type]” rather than manually defined. If the reaction behavior defined in the reaction entity has multiple paths, then parameters should be set in the entity’s task node to determine the proper path.

While not strictly dictated by the language, by convention the target node of a reaction transition should be a multitask node, even if the desired reaction behavior consists of only one atomic task. In this manner contingency activities can easily be abstracted. The destination nodes of the reaction transitions in Figure 20 follow this convention.

4.4 Concrete Entity Binding

The abstract entities defined in the abstract entity section must be bound to specific entity values for a scenario to be adapted for a specific situation. This allows the generic scenario to be converted to a form closer to a usable one for a simulation system for an execution run.

All concrete entities must correspond to a defined abstract entity or entity set. In this manner, the entity receives its behavior descriptions. The concrete entity additionally is given the specific simulation run parameter values. These values must conform to the constraints as provided in the abstract entity section.

The scenario developer would select an abstract entity to transform into a concrete entity. The user may provide the concrete entity with a unique identifier or the system can generate one automatically. Note that this identifier is not necessarily the same as the

identifier of the abstract entity; in the latter, the entity within its role is stressed rather than the actual entity modeled in concrete form. The system then displays the list of attributes associated with the entity for the developer to select the appropriate value for the scenario. The value must satisfy the relevant constraints as defined in Section 4.2.1. These values make up the initial parameters of the entity in the scenario. Additionally, abstract entity sets must have a value for the quantity of concrete entities in the scenario. For example, if an abstract entity set has a cardinality of 0..4, the user must specify whether there are 0, 1, 2, 3 or 4 concrete entities in the scenario. The user may set the other parameters of the entities created from the abstract entity set individually or as a block.

Once the entities involved have all been represented with concrete entities, the tool would then allow the user to save the instantiated scenario in a file. This file would be processed by a translator program which converts it into a simulation-specific scenario input file. Due to time and resource constraints, the translation programs have not been created.

4.5 Summary

The visual scenario language and tool allow the scenario developer the ability to design a scenario with players—entities that are participants in the scenario—and player mission behavior foremost in mind. By creating scenarios using the abstract entity and its extension, the abstract entity set, the developer can focus on roles as well as structure.

Adopting the Scenario-Based Specification Diagram (SBSD) language as the description of entity mission behavior takes advantage of the results of previous research

done at AFIT. The SBSB has an easy to understand and use syntax that makes it ideal to describe behaviors within a scenario.

Finally, the concrete entity binding methodology allows the creation of entities for use within a scenario, fully parameterized. Translator programs would map the instantiated scenario into a simulation system specific form.

The next chapter describes the specific implementation of the tool as well as results from using the language and tool in composing scenarios.

V. Implementation and Use Cases of the Visual Language

5.1 Introduction

The language described in Chapter 4 was implemented into a software tool using the Java programming language. The visual language tool provides a graphical user interface (GUI) that allows users to construct the entities participating in a scenario beginning with an abstract form along with their associated behaviors in the Scenario-Based Specification Diagram (SBSD) language. This chapter covers the details of the implementation program in both of its aspects: the GUI and the data storage and retrieval component. The chapter also demonstrates some example scenarios generated from the language using the tool and how they map to a conceptual scenario form focusing on items of interest to a scenario developer.

5.2 Implementation of the Visual Language Tool

The Defense Modeling and Simulation Office (DMSO) held a meeting in May 2003 where five shortfalls in the realm of simulation systems were identified. The second shortfall is described as:

The relationships between verb/task oriented operational views and noun/entity oriented system views are undefined in DoD Framework, in Unified Modeling Language, and are implicit tribal knowledge in the Military Decision Making Process. Framework should define the regular expression grammar for specifying these relationships. (Unambiguously machine parsability issue) [Personal correspondence from DMSO, May 2003]

The complete text of the memo is provided in Appendix A.

The visual language is an attempt to explore possible solution spaces for this problem. The tool implements the language and adds visual presentation aspects to it.

The scenario language itself, without the presentation component, can be readily defined in Backus-Naur Form; the rules of production are listed in Appendix B, Tables 1 through 3. The BNF parsing rules also provide a guide to the development of the storage and retrieval of the abstract scenarios produced by the tool. Details of the data storage component are provided in Section 5.2.2.

Figure 21 shows a partial UML class diagram of the GUI portion of the language tool. In order to allow the reader to see the basic structure of the classes and their associations, several details have been omitted from the diagram. Only a few key attributes of some classes are shown; all methods are hidden from view. Many associations are also hidden.

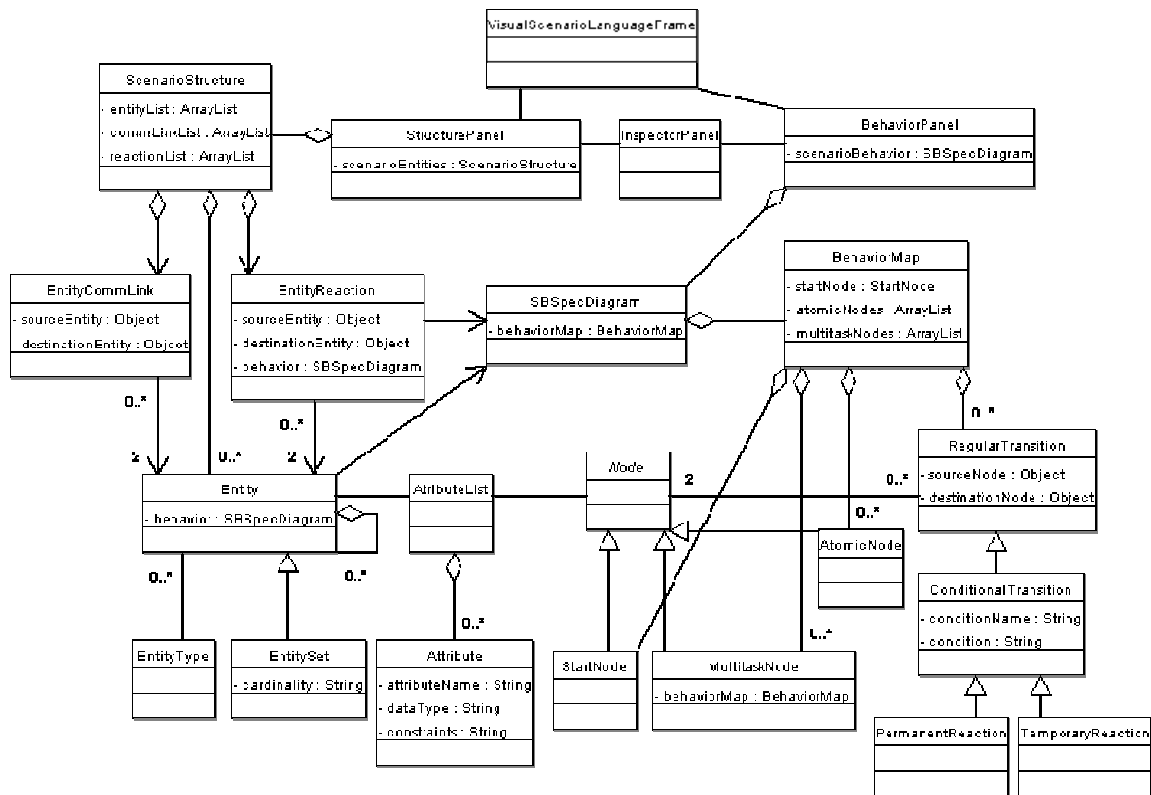


Figure 21. UML Class Diagram of the Visual Tool Program GUI Component

The class that provides the “glue” upon which the program’s GUI hangs together is the VisualScenarioLanguageFrame class, shown centered at the top of Figure 21. All of the GUI support class instances are initialized in this class: the menu, tool bars, drawing panels and inspector panels. Figure 22 shows a screenshot of the tool running.

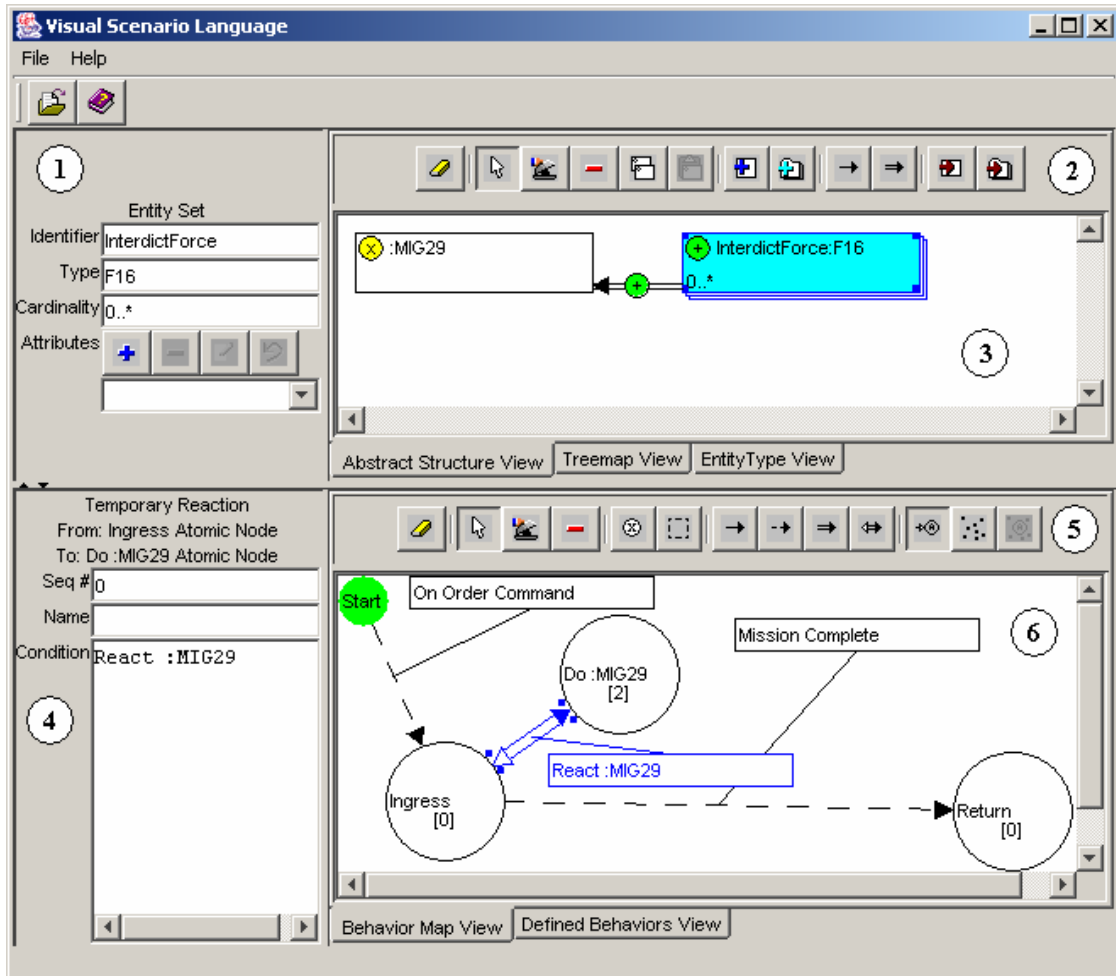


Figure 22. Visual Scenario Language Tool Screenshot

5.2.1 Graphical User Interface Component

The software tool, when running, is divided into various panels to provide the scenario designer access to components of interest. The tool has conventional

Windows™ components—the menu bar and traditional toolbar—as well as application-specific panels. The remainder of the discussion will focus on these panels.

Items 1, 2 and 3 in Figure 22 describe the entity-centric view. Item 1 is the inspector panel of the selected entity component; in this view, the selected entity is the abstract entity set `InterdictForce`. The inspector panel provides the fields for the user to enter relevant data with respect to the selected component.

Item 2 is the Abstract Structure View’s toolbar. Using the buttons on this toolbar, the user may (from left to right): clear the scenario; select a component; hide/show details of the selected component; delete the selected component; copy the selected entity-type component; paste a clone of the copied entity-type component; create an abstract entity; create an abstract entity set; create a communication link; create a reaction transition; convert the selected entity set to an entity; and convert the selected entity to an entity set.

Item 3 is the display panel for the Abstract Structure View. As shown in the figure, the entities and transitions are shown and manipulated in this panel. The components in this panel conform to the descriptions in Section 4.2. Additionally, to distinguish the currently selected component from the others in the panel, corner “handles” and color are used. The selected entity/entity set is cyan with a blue outline; the four blue handles are in the entity’s corners. In the case of a selected transition (reaction or communication link), the transition is blue with blue handles defining the “click-selectable” area of the transition.

The Treemap View (the first unselected tab in Item 3) is where the “concretization” of the abstract scenario takes place. The Entity Type View (the second unselected tab) is the entity type editor pane, covered in greater detail in Section 5.2.1.1.

Items 4, 5 and 6 describe the behavior-centric view of the selected entity view component. If there is no selected component or the selected component cannot have a behavior (i.e., communication link or reaction-only entity, such as MIG29 in the figure), the panels involved here are inactive.

Item 4 is the inspector panel for the behavior. Like the inspector panel for the abstract entity, it is used to allow the scenario developer to view and update data with respect to the selected behavior component.

Item 5 is the behavior toolbar. These options are: clear the behavior panel (effectively removing the behavior tasking from the selected entity); select behavior component; hide/show the details of the selected multitask component; delete the selected behavior component (task or transition); create an atomic task node; create a multitask node; create a regular transition; create a conditional transition; create a permanent reaction transition; create a temporary reaction transition; select behavior full view; select behavior small view; select behavior fisheye view. These last three merit some discussion. The full view is displayed in Figure 22. In it the components are “full sized” with the details viewable. It is possible to create a large behavior task for an entity and in this case it may be impossible to view the entire behavior map. The small view collapses the size of the components to allow the developer an overview of the entire behavior map. However, the labels are not displayed in the small view. The fisheye view is a

compromise where those items not under the mouse cursor are in a small view while the task items under the cursor are in full view. Task items immediately connected to the full size task node are also full size. Unfortunately, this feature was not completely implemented due to time constraints, and the option is disabled.

Item 6 is the display panel for the behavior map. It consists of two panels supporting two views. One is the Defined Behaviors View panel; this panel is where the permitted atomic tasks of an entity type are defined. The other is the Behavior Map View panel (displayed in the figure); the components in this panel conform to the descriptions given in Section 4.3. Like the entity panel, there is an extension to highlight for the developer the selected component: blue outline with blue corner handles; the task “Return to Base” is the selected component in the figure.

In defining a scenario, the developer must first consider what entities are involved. These entities are of certain types and would have behavior characteristics and constraints by nature of their type. The next section covers the entity type and its associated editor within the tool.

5.2.1.1 Creating Entity Types and Subtypes

The purpose of the entity type is to provide the scenario developer a ready-made set of attributes (modifiable parameters) and associated constraints from which scenario entities may be created; i.e., the entity type provides a means of describing the capabilities of an entity of the specified type. Additionally, as entities may react generally to other entities of a certain type, the type provides a mechanism to identify entities by type such that appropriate reaction behaviors can be readily defined.

Entity types naturally lend themselves to a hierarchical classification of greater specialization; thus, a means to capture and reuse these attributes would greatly simplify the work of the scenario developer. For a detailed discussion on the concept of entity types, refer to Section 4.2.1.

In order to support the form of inheritance with respect to entity types, the tool provides an entity type editor as part of the scenario structure pane, selected with the Entity Type View tab. Figure 23 shows the entity type editor in the active window pane.

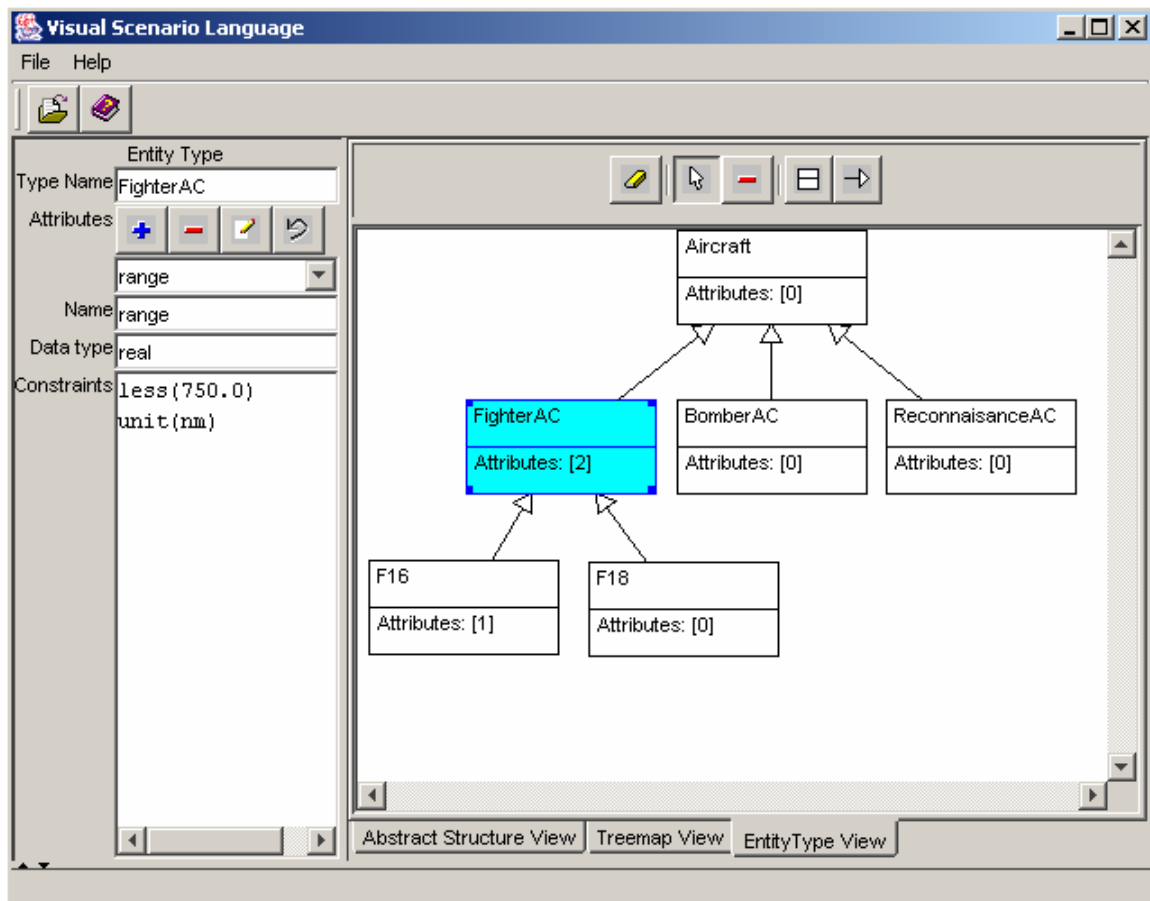


Figure 23. Entity Type editor panel in VSL tool

The editor is based on the notation of the Unified Modeling Language (UML) as determined in Section 4.2.1. The *entity types* are represented by the boxes, with an entity type name in the upper portion of the box. The lower portion indicates the number of attributes that the entity type is directly responsible for introducing into the inheritance chain; i.e., it does not reflect the number of attributes that it inherits from its supertype and higher levels of abstraction or *generalization*.

The arrows represent the immediate generalization of the entity type. The arrowhead is associated with the generalized, or *parent*, entity type. An entity type may have only one immediate supertype. Entity types without a supertype are considered *base entity types*. The inheritance chain may not have cycles; i.e., an entity type may not be its own ancestor irrespective of the number of intervening generations. Allowing such a structure would defeat the purpose of inheritance: in a cycle, it would be impossible to determine which entity type is the supertype and which the subtype. Indeed, in a cycle all types are supertypes and all types are subtypes. The inheritance chain would take on an M. C. Escher-like surreal quality.

Attributes can be added or removed from the subtypes through the inspector panel on the left side of the entity type pane. The attributes in the entity type represent the variables with associated constraints that apply to an instance of the entity type; i.e., an entity of the entity type in a simulation scenario would potentially have as parameters the attributes whose values must comply with the constraints contained within the attribute. Removing an attribute from an entity type's inherited attribute list is equivalent to stating that the attribute in question is not applicable to an entity of the subtype. Attribute

removals are also inherited by subtypes: if a supertype, containing attribute A in its list, has a subtype which has A removed from its list, then subtypes of that subtype do not inherit attribute A.

Constraints may also be added or removed from any given attribute within a subtype. Subtypes derived from the changed entity type also inherit these changes. For example, an entity type contains an attribute B with a list of values as the set of constraints for any assigned value for that attribute. A subtype is subsequently created from that type; the subtype inherits attribute B with the constraints. The user may change the constraints in the subtype's copy of attribute B without affecting the supertype's copy of the attribute. Any subtype derived from this subtype will reflect the changed version of constraints for attribute B.

Entity types may be deleted if they do not have dependent subtypes. There is one exception to this rule: if an entity type does not alter in any way its inheritance chain (i.e., it does not delete or alter any inherited attribute, nor add any new attribute), then it may be deleted and the dependent subtypes will be associated with the now-deleted entity type's immediate supertype.

The toolbar for the entity type editor provides (from left to right in Figure 23): clear the entity type panel, select an entity type in the panel, delete the selected entity type, create a new entity type, associate the inheritance link between the subtype (source) to the supertype (destination).

As the developer creates entity types in the Entity Type View panel, the developer may also create in the Defined Behaviors View panel those valid atomic behavior tasks

frames that an entity of the entity type may perform. The tasks that the developer creates in this panel are the only atomic tasks that an entity of this entity type may perform.

When the developer is creating the abstract scenario and in the process of assigning a behavior map to an abstract entity in the scenario, the tasks that are created as a part of the behavior map must be from the defined behaviors.

Entity type diagrams may be saved to and loaded from disk file for later editing and reuse. Multiple entity type diagrams may be created for different applications or different simulation systems as necessary. In any case, once entity types are defined, the developer may proceed to creating entities and their behaviors within the scenario.

5.2.1.2 Creating Entities

Once the entity types are defined, the scenario developer may begin creating an abstract scenario. As discussed in Section 4.2, the heart of a scenario is the set of entities playing specified roles. The entity type prescribes the limits upon which an entity of the type may perform. An entity operating in a specific role may have additional constraints beyond those inherent within the nature of the entity itself. This is the purpose of the abstract entity: to define the additional constraints and behaviors of an entity playing a role within a scenario.

The developer defines the abstract scenario within the Abstract Structure View panel of the tool. First the entities playing roles in the scenario must be entered into the panel. The developer does this by determining whether an abstract entity or entity set is the most appropriate for the role. As discussed in Section 4.2.2, the only difference between the abstract entity and abstract entity set is that the set has cardinality in addition

to the normal entity features. If the role is played by multiple instances of a type performing the same behavior, then the abstract entity set is the appropriate representation of the player.

To create an abstract entity or entity set, the developer selects the corresponding button on the Abstract Structure View toolbar. The developer then places the mouse cursor over the desired location of the entity on the display panel. The developer clicks the left mouse button and the entity (or entity set) is created with its upper left corner at the mouse cursor. If the point is inside another entity, the created entity becomes a subcomponent of that entity. In this way nesting is accomplished, as defined in Section 4.2.3. The entity inspector panel is refreshed with the information of the newly created entity, allowing the developer to enter the appropriate data for the entity. This data includes the identifier, the entity type and the cardinality, if the created entity is an entity set. Additional constraints imposed by the role the entity is to perform in the scenario beyond those due to the nature of the entity (as defined in the entity type) are also entered in this panel.

If no identifier is supplied, then the created entity is a *reaction entity* as described in Section 4.2.2. The entity of type MIG29 in Figure 22 is a reaction entity. Normal entities may have associated behavior, contain nested entities, have communication links with other entities, and may have reaction transitions to any type of entity (including reaction entities), but reaction entities may only be the destination of a reaction transition. The tool enforces these specifications.

5.2.1.3 Creating Entity Transitions

The developer may define a communication link between non-reaction entities. Links are considered bidirectional channels by the language. The output language description of the scenario has a “comm_link” clause for both entities referring to the other entity regardless of which is the source or destination of the transition. Nested entities are assumed to have communication links up and down the nesting and are therefore not required—neither does the tool permit the developer to create an explicit link up or down the nesting hierarchy. However, such links are not automatic for sibling entities; i.e., entities that are nested within the same entity but not nested to each other. If a communication link between such entities is required by the scenario, the developer must explicitly create it using the Create Communication Link button in the Abstract Structure toolbar.

Unlike communication links, reaction transitions are unidirectional. The source entity must be a proper entity; i.e., not a reaction entity. The destination entity may be either a proper entity or a reaction entity, although the typical case is a reaction entity; the reasons for this are discussed in Section 4.2.4. Reaction entities may only be the destination of a reaction transition, never the source.

For each non-reaction entity and reaction transition in the scenario, a behavior may be composed. This is done in the Behavior panel and is the next topic.

5.2.1.4 Creating Behaviors

The purpose of a scenario is to describe the players and their behaviors in a manner that allows a simulation system to execute the scenario to provide outcome data

that ultimately aid commanders make decisions in real situations. The purpose of a given simulation system may be training such that commanders will be better prepared to face these real situations. The purpose of another system may be to provide a commander the probable results of a course of action. But regardless of the purpose of a specific simulation system, the players in the scenario must execute some behavior. The modeled players—or entities—must have the behavior defined so that they can perform within the running simulation. This section describes how behaviors for entities in the abstract scenario system are created.

The bottom panel of the tool is the behavior editor. It consists of the inspector panel to the left, and the behavior map with the behavior toolbar to the right. The developer determines the behavior of the currently selected proper entity, entity set or reaction transition. The tool will not allow a behavior map to be created for reaction entities. The developer places atomic nodes on the map by selecting the Create Atomic Node button and clicking on the behavior map panel at the desired location. Atomic nodes represent the tasks the entity will potentially perform during the course of a simulation run. The developer names these tasks and associates attributes, which are placeholders for setting parameter values upon entry to the task during the execution of the simulation. Access to these variables is through the inspector panel which is context sensitive to the currently selected component in the behavior map.

The developer may place multitask nodes on the behavior map in a similar manner using the Create Multitask Node button. Multitask nodes do not represent tasks in and of themselves, but rather represent a “macro-task;” i.e., a behavior submap that can

be considered as a single task. Multitask nodes do not have parameters associated with them directly, but the atomic nodes that may be placed in them do. Both types of nodes are discussed in detail in Section 4.3.1.

Once a set of nodes are placed on the behavior map, the developer creates transitions to the nodes. The behavior map upon creation contains one start node, identified as a slightly smaller, green circle labeled “Start.” In order for a task to be executed, there must be a path from the start node to the node representing the task. The tool identifies those nodes that are not accessible to the start node by drawing them in red. Cycles are not permitted in the behavior map; allowing them could conceivably lead to endless loops during the execution of the simulation. This is enforced within the tool by not allowing a node to be the destination of more than one transition.

The transitions are of the four types defined in Section 4.3.2. The reaction-type transitions are only permitted at the highest level of the behavior map; i.e., they are not permitted in submaps within multitask nodes. A node may be the source of multiple transitions, of which only one may be a regular transition. The other transitions must be of a form of conditional transition with a guard condition. The purpose is to allow a task to be prematurely exited if the guard condition evaluates to true; otherwise, upon normal completion of the task, the regular transition is traversed. Like all other user entries, the data for transitions (sequence number, name, guard condition) are accessed through the inspector panel.

5.2.1.4 Reusing Abstract Entities and Behaviors

The tool provides a means of reusing an entity within a scenario as another component with the same or slightly different behavior mission. This is performed by the copy/paste entity buttons in the Abstract Structure View toolbar. The entity produced is a clone of the original and the developer may modify the internal nested entity structure and associated behavior map without affecting the original entity's structure or behavior.

The cloned entities and behaviors are manipulated in the same manner as entities and behaviors created directly by the scenario developer.

5.2.1.5 Converting the Abstract Scenario to Concrete Scenario

Once an abstract scenario has been defined, a concrete scenario may be generated. This is performed in the TreemapView panel. The panel automatically generates a treemap view from the abstract scenario upon entry to the panel. The concrete scenario requires some specific detail that is not essential to the abstract form of the scenario; e.g., the specific number of entities represented by an entity set. The concrete scenario automatically places an entity set "shell frame" around entities derived from the entity set with at least one entity as a placeholder; if the entity set's cardinality has a minimum value greater than one, then that minimum number of entities will be generated in the shell automatically. If the cardinality allows zero entities, then the developer may delete the one entity in the shell frame. The developer also enters the starting parameter values of the entity attributes in this view panel.

When the developer has resolved all of the detail issues, the concrete scenario may be saved to a disk file. The software tool does not support reading concrete scenario

files at this time. The output concrete scenario file would subsequently be used as input by a scenario translation program which maps the entities, behavior task frames and parameter values into a scenario file format compatible with the target simulation system. As discussed previously, a translation program was not developed as a part of this research due to time constraints.

5.2.2 Storage and Retrieval of Data Components

The tool stores the various artifacts of the system in several files for reuse. The abstract scenario is saved in a VSL file which contains the involved entity definitions, links and reactions as well as their associated behaviors. The entity type table is saved in a VST file. The concretized scenario is saved in a VSC file.

The advantage of this approach is that different entity type tables can be custom created to mirror the capabilities and expectation of specific simulation systems that are not completely compatible conceptually with each other. However, for those systems that are conceptually congruent (i.e., the type of entities, the user-selectable attributes and permissible behaviors of the entity types are essentially the same between these compatible systems), this scenario development tool provides a framework for reusing scenarios in the different simulation systems. The translation program could be designed to handle conversions of near types; e.g., if one system may use a floating point number for a given attribute where another system uses an integer then the translation program would have a conversion function to allow the scenario data to be transformed into a format compatible with the other's format.

Another advantage of the structure is that it allows the use of XML for storage, search and retrieval. It is possible to transform the abstract scenario and entity type files into XML and subsequently use modern XML tools to store the information into a database, leveraging advanced database systems' capabilities for search and retrieval. The scenario language design was mapped to a parsable BNF notation to facilitate this transformation of the libraries of entities, types, tasks, attributes and scenarios from their native format into XML for use in other systems. But at this point, the system provides a simple save and load capability to the abstract scenario file and the entity type library file.

5.3 Example Abstract Scenario Design in the Language Tool

The tool was used to create the simple scenario involving two commands: a Soviet opposing force and a US distinguished force. The USSR force consisted of a MIG29 fixed wing aircraft and a Mi24 rotary wing aircraft. The US force consisted of F14D and AH64A aircraft. Figure 24 shows the scenario with the AH64A entity in focus. Appendix C contains the abstract scenario VSL file to show the conversion from visual representation to a context free grammar.

The simple scenario was selected for examination because it was provided by the OneSAF simulation system as an introductory scenario. In this case, the scenario describes the planned mission. The behavior map in the figure corresponds to the tasking for the US forces AH64A aircraft. The tasks shown represent the task frames developed by the OneSAF team that are permissible for entities of the fixed-wing aircraft (FWA) and rotary-wing aircraft (RWA) entity types. Each of these task frames contain individual tasks that are not directly manipulable by the user of the OneSAF system.

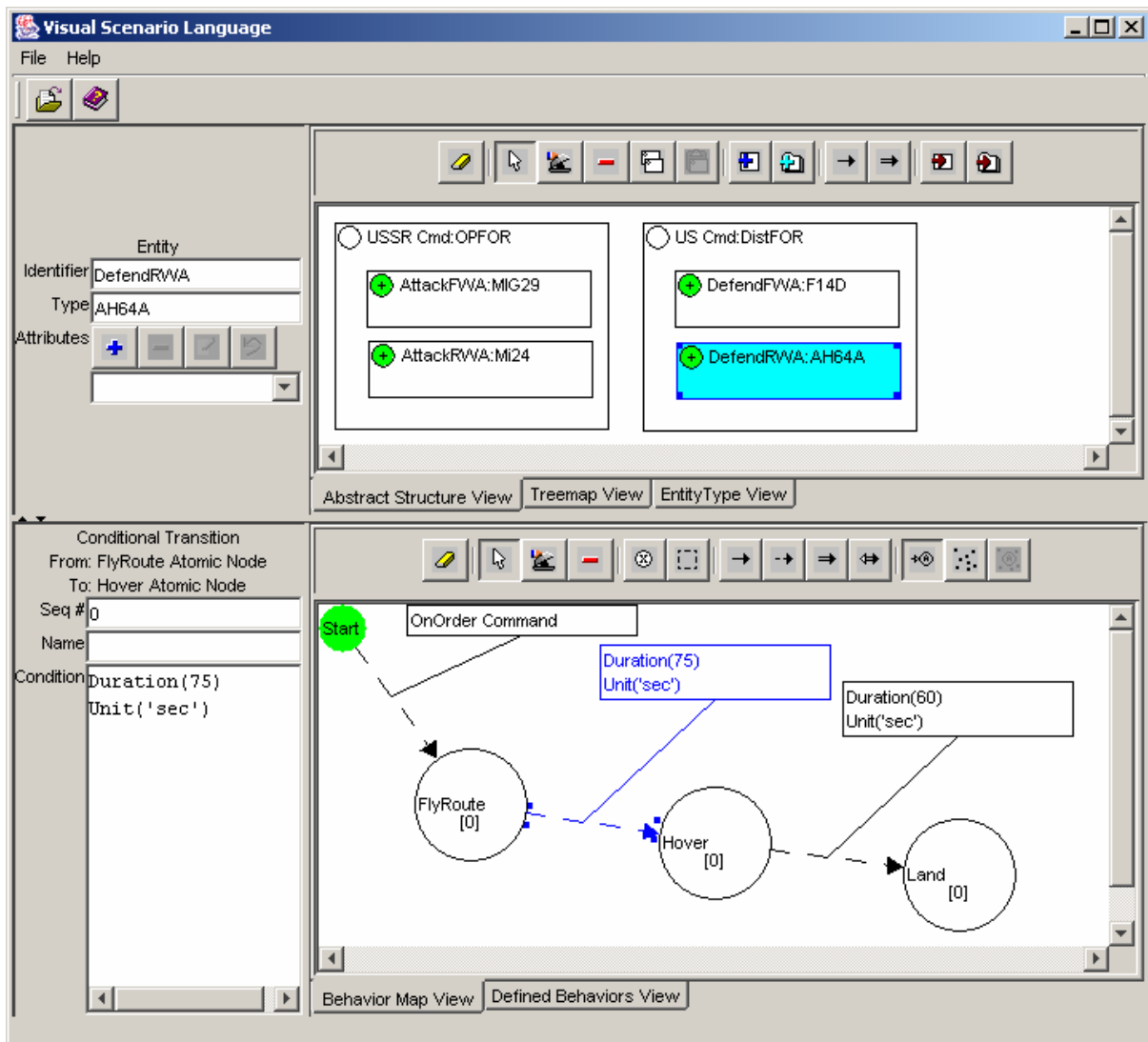


Figure 24. Simple Scenario in VSL

While this represents the minimal level of granularity for the simulation user, a scenario developer may require a greater degree of detail. The Defined Behaviors View allows the scenario developer the ability to view these task frames in greater detail and to set the tasks within the task frame.

An example of this detail is in the diagram shown in Figure 25. The task frame of Figure 24 is expanded to show the detailed tasks within the Fly Route task frame. It also

shows the reactions that are possible to an entity executing this task frame as part of its scenario in a simulation run.

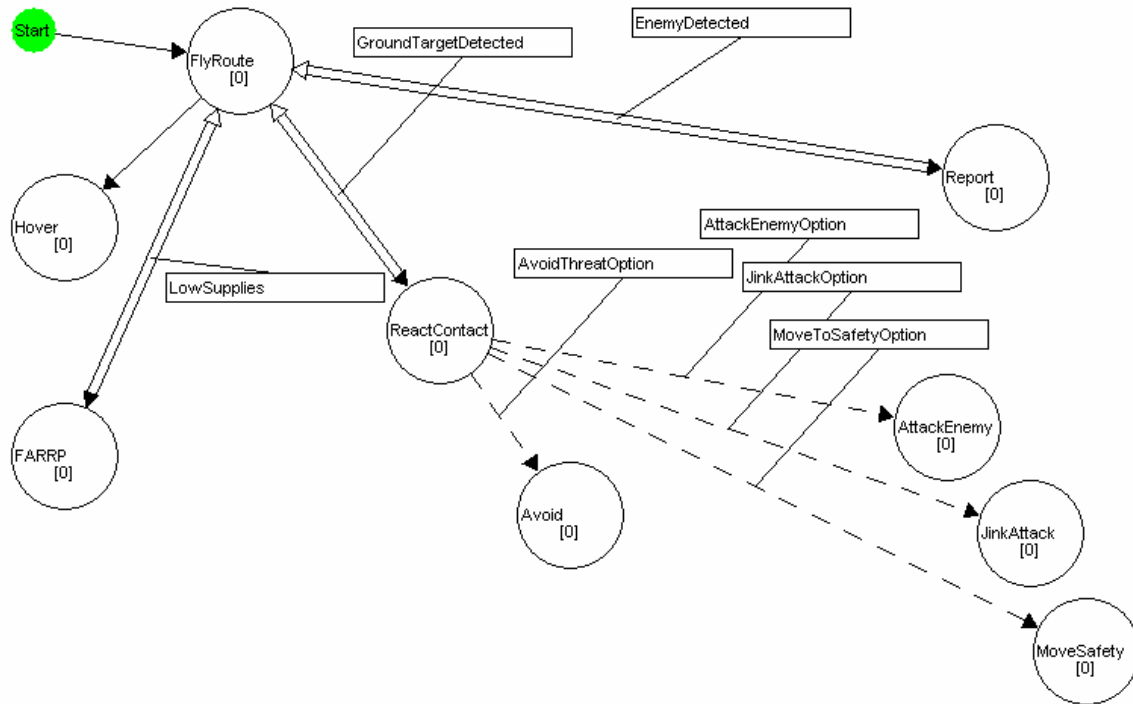


Figure 25. Detailed View of Fly Route Task Frame

The visual language and tool thus provide enough expressiveness to describe scenarios in an abstract form and yet provide some degree of detail. With further development, it is possible to refine the tool to produce output directly useful to simulation systems; i.e., parsable in a format compatible with the simulation system.

5.4 Summary

The visual language provides a means of expressing scenarios in terms common to all simulation systems: entities (actors or players in the scenario) and behaviors those entities perform in the simulation. The tool implements and enforces the constraints of

the visual language. These constraints can be described using a context free grammar in Backus-Naur Form. Additionally, the constraints of the simulation system whose scenario composition capabilities are modeled with the entity type library system are also enforced. While subtle, this is a crucial value added element of this work.

This language and tool thus provide a basic framework, when coupled with translation programs, by which scenarios can be composed by developers possessing knowledge of the problem domain, without necessarily having detailed knowledge of the esoteric features of the simulation system's language. The translation programs provide the mapping from the entities and behaviors described in the scenario to the form accepted by the simulation system. While the translation was not accomplished with this research, it provides the next logical step in advancing the state of generic scenario composition to live simulation systems.

VI. Conclusions and Recommendations

6.1 Introduction

The visual scenario language proposed in this thesis has potential to aid the rapid development of scenarios by problem domain experts. With the entities and tasks constrained by the entity type dictionary, developers can assemble task frames and create complete scenario missions involving many entities. The scenarios produced by the current version of the system are useful in conceptualizing scenarios, allowing scenario developers visualize how a scenario might be constructed for a live simulation system. Eventually, it may be possible to construct the live system scenarios directly using future versions of the tool.

6.2 Conclusions of Research

The visual language is simple, easy to learn and use. It uses a basic notation that can be drawn by hand to sketch out a scenario concept, while the tool adds a constraint checking mechanism. The visual language maps to a language described with a context free grammar that can be easily parsed, aiding constraint checking of the scenario. These fulfill the criteria of reliability, translatability and reduced the frequency of errors; i.e., detected and displayed syntax errors. The environmental provision of the entity type with its associated defined behavior description helped fulfill the criterion of reuse. As all components of a scenario are defined (entity, behavior, constraints and parameters) and interrelated, the expressiveness of the language is adequate to describe scenarios in a near-simulation format.

The visual nature of the language allows the developer to see scenarios in a more intuitive form, abstracting away many of the details, but providing access to them through inspector panels. This results in an increase in the understandability of the scenario, particularly for individuals not familiar with the syntax of scenario description languages of particular simulation systems.

The visual language and tool are presently at a very basic point; more work is necessary to refine the language tool and adapt it for use with live simulation systems. The essential components are present: entities that operate within the scenario; the elements describing the behavior (tasks and transitions); and attributes of the entities (constraints on the entity inherent to its nature and additional constraints due to the requirements of the role the entity plays within the scenario). What remains is mapping the components of the visual language from its textualized form to the format of a specific simulation system's scenario file.

6.3 Significance of Research

This research demonstrates that a general-purpose visual language can express relations between entities and their behaviors in a way that allows the scenario developer to have a higher understanding of the scenario, seeing it in visual terms rather than in voluminous script files—typically the form in which scenarios are built and stored.

Simulation systems are increasingly complex; scenarios are likewise growing in complexity. The ability to abstract away much of the detail can only help developers maintain control over the increased complexity. A graphical user interface additionally allows a user of the visual tool a rapid development approach to scenario construction.

The abstract nature of the language coupled with the GUI thus tends to not only aid understanding of complex scenarios, but also eventually promotes the ability of users to define scenarios without direct intimate knowledge of the simulation language format.

6.4 Recommendations for Action

For a proof-of-concept, at least one translation program to convert an output concrete scenario file into a compatible scenario file for use in a functional simulation system should be developed. Additionally, translation programs may be developed to load existing simulation systems' scenario files to extract the entity types, associated behavior tasks and developer-accessible parameters into an entity type definition file. In this way, existing scenarios can be leveraged to create a dictionary of "legal entity types" from which scenario composers can create abstract scenarios.

A conversion utility should also be developed to convert the abstract scenario and the entity type library files to XML take advantage of database systems as alluded to in Section 5.2.2.

6.5 Recommendations for Future Research

It appears that the ADORA approach to software engineering has advantages that are applicable to engineering simulation scenarios. The ADORA project is an on-going research effort at the University of Zurich; it is by no means a completed system or methodology. As the state of research progresses, it has the potential to provide an improved means of modeling simulation scenarios. Although ADORA was not itself directly adopted in this research, it may well progress to a point where it could directly apply to the simulation community as a general purpose modeling language. It is

strongly recommended that developments in the ADORA project be kept in view while continuing improved means of describing, defining and implementing scenarios.

Another area of research that would be beneficial is the use of visual languages in computer programming. While this is a more pragmatic approach than theoretical—it tends to emphasize a product that may by its nature be limited to a particular simulation system—it still would offer benefits to the simulation community. Developing operational scenarios quickly and easily while controlling the complexity by a Visual Simulation for Windows language would likely be a welcome addition to the simulation and modeling community.

6.6 Summary

As it follows up on past efforts, this work represents the completion of one phase of research in this arena. It also marks the beginning of the next phase: more work will need to be done in order to produce tools that can aid scenario developers rapidly create scenarios for simulation systems to meet the demand for high-quality simulations for new tactics, training and analysis. The visual language provides one more element towards achieving that goal.

Appendix A

May 200

Context:

Enable DoD transformation from a Forces-based, materiel-centric (focused on the "players") Cold War posture to a Mission-based, capability-centric (focused on the "playbook") asymmetric warfare focus.

Need:

A disciplined, repeatable procedure for explicitly specifying the mission and assessing mission accomplishment. Mission statement and assessment procedure must account for the tangible, physical objectively measurable factors (traditional T&E) as well as the intangible, cognitive ultimately subjective factors (traditional warfighter expertise) that constitutes mission success.

Requires:

Four things: (1) a framework with supporting methodology, (2) a critical mass of end user content, (3) a business model with associated program elements and release policies, and (4) tools and utilities. (1) and (3) are Joint/OSD roles. Per the Aldrich devolvment memorandum, (2) and (4) are Service or Service as a Joint Executive Agent responsibility. (2) and (4) are a waste of effort in the absence of (1) and (3).

Recent Accomplishments:

Army has been most significant customer: Battle drills in use by CFLCC in Kuwait, UOB and EOB data loads for 1st Cav and 4th ID, C3 Driver certification of ABCS 6.3, Comanche AoA, Future Combat Systems mission decomposition included in O&O, ORD, C4ISR AoA, and T&E strategy.

Other Service and Joint collaborations include: JSF Authoritative Modeling Information System (organizing principle for data warehouse), OSD P&R Training Transformation (mission-based readiness assessment), Air Force Operation Desert Shift (reduce AOC footprint by 50%), Navy Multi-Mission Aircraft (MMA mission decomposition).

Remaining Shortfalls:

(1) Decomposition yield an explosion of detail. Need framework for combining decomposition detail into actionable information. (Complexity issue).

→ (2) The relationships between verb/task oriented operational views and noun/entity oriented systems views are undefined in DoD Framework, in Unified Modeling Language, and are implicit tribal knowledge in the Military Decision Making Process. Framework should define the regular expression grammar for specifying these relationships. (Unambiguously machine parsability issue)

(3) Emerging operational forces need to re-plan, re-constitute on the fly. Need framework for explicit composable interfaces between verb/task/operation specification and noun/entity/system specification. (adaptability issue)

(4) Operation/System specifications are universally late to need. Need framework to objectively distinguish between core, common, and custom mission content. Framework then enables a business model (3rd thing above) to ensure critical content (2nd thing above) is available on the required timeline. (timeliness, cost issue)

(5) The leap from performance measurement (in MKS units) to mission utility is still a subjective judgment. Need framework for objectively structuring and stating the subjective assessment. (quantifying mission demand issue).

Börland "together Soft" - a better version for UML than Robt/ Rose. GK

Figure 26. Personal correspondence from the Defense Modeling and Simulation Office (DSMO), May 2003

Appendix B

Table 2. Visual Language in BNF (basic definitions)

<code><scenario> ::= scenario [name <identifier>] "(" <entity_clause> {<entity_clause>} ")"</code>
<code><attribute_constraint_clause> ::= attribute "(" <identifier> type ""<type>"" "(" constraint ""<constraint>"" {constraint ""<constraint>""} ")" "</code>
<code><attribute_list> ::= attribute_list "(" <attribute_constraint_clause> {<attribute_constraint_clause>} ")"</code>
<code><boolean_expression> ::= ***</code> Note: this is a placeholder for conventional Boolean expressions rather than an explicit definition
<code><constraint> ::= <value> <range></code>
<code><identifier> ::= (""<letter> { <letter> <digit> }"" unnamed)</code>
<code><identifier_clause> ::= identifier <identifier></code>
<code><maximum_value> ::= <value></code>
<code><minimum_value> ::= <value></code>
<code><natural_number> ::= <digit> {<digit>}</code>
<code><range> ::= within <minimum_value> to <maximum_value></code> Note: An additional constraint is that maximum value must be equal to or greater than minimum value
<code><transient_clause> ::= transient <float> <float></code>
<code><type> ::= boolean char double float int real string</code> Note: These are basic types for use in the parameter value setting. It may not be a complete list for the specific simulator system and some forms may not be supported by a given simulation system.
<code><value> ::= <boolean_value> <char_value> <double_value> <float_value> <int_value> <real_value> <string_value></code>

Table 3. Visual Language in BNF (abstract entity definitions)

<code><abstract_entity> ::= entity "(" <entity_id_clause> <transient_clause> <abstract_entity_clause> ")"</code>
<code><abstract_entity_clause> ::= <identifier_clause> <entity_type_clause> [<attribute_list>] {<entity_comm_link_clause>} {<entity_reaction_clause>} [<behavior_clause>] {<inner_entity>}</code>
<code><abstract_entity_set> ::= entity_set "(" <entity_id_clause> <transient_clause> <cardinality> <abstract_entity_clause> ")"</code>
<code><cardinality> ::= cardinality ""<minimum_cardinality>..<maximum_cardinality>""</code> Note: An additional constraint is that maximum cardinality must be equal to or greater than minimum cardinality
<code><entity_clause> ::= (<reaction_entity> <reaction_entity_set> <inner_entity>) {<entity_clause>}</code>
<code><entity_comm_link_clause> ::= comm_link <entity_id_clause></code>
<code><entity_id_clause> ::= entity_id <natural_number></code>
<code><entity_reaction_clause> ::= reacts_to <entity_id_clause> [executing "(" <behavior_clause> ")"]</code>
<code><entity_type_clause> ::= entity_type <identifier></code>
<code><inner_entity> ::= (<abstract_entity> <abstract_entity_set>)</code>
<code><maximum_cardinality> ::= <natural_number> *</code>
<code><minimum_cardinality> ::= <natural_number></code>
<code><reaction_entity> ::= reaction_entity "(" <entity_id_clause> <entity_type_clause> ")"</code>
<code><reaction_entity_set> ::= reaction_entity_set "(" <entity_id_clause> <cardinality> <entity_type_clause> ")"</code>

Table 4. Visual Language in BNF (behavior definitions)

<code><atomic_node> ::= <basic_atomic_node> ([<regular_transition_statement>] {<conditional_transition_statement> <permanent_reaction_statement> <temporary_reaction_statement>})</code>
<code><attribute_clause> ::= attribute <identifier> value ""<value>""</code>
<code><basic_atomic_node> ::= <node_id_clause> <transient_clause> <node_identifier_clause> {<attribute_clause>}</code>
<code><basic_multitask_node> ::= multitask_node <node_id_clause> <transient_clause> <node_identifier_clause> contains "(" {<inner_node>} ")"</code>
<code><behavior> ::= start_node "(" <node_id_clause> <transient_clause> (<regular_transition_statement> <conditional_transition_statement> <unreachable_node>) {<conditional_transition_statement> <unreachable_node>} ")"</code>
<code><behavior_clause> ::= behavior "(" <behavior> ")"</code>
<code><conditional_transition_statement> ::= conditional_transition "(" <guard_clause> <node> ")"</code>
<code><guard_clause> ::= guard (""<boolean_expression>"" undefined) [guard_identifier <identifier>] [guard_sequence <natural_number>]</code>
<code><inner_atomic_node> ::= <basic_atomic_node> ([<inner_regular_transition_statement>] {<inner_conditional_transition_statement>})</code>
<code><inner_conditional_transition_statement> ::= conditional_transition "(" <guard_clause> <inner_node> ")"</code>
<code><inner_node> ::= <inner_atomic_node> <multitask_node> <inner_unreachable_node></code>
<code><inner_regular_transition_statement> ::= regular_transition "(" <inner_node> ")"</code>
<code><inner_unreachable_clause> ::= conditional_transition "(" <guard_clause> <node_id_clause> ")"</code>
<code><inner_unreachable_node> ::= unreachable "(" (<basic_multitask_node> <basic_atomic_node>) [<unreachable_reference_clause>] {<inner_unreachable_clause>} ")"</code>
<code><multitask_node> ::= <basic_multitask_node> [<regular_transition_statement>]</code>
<code><node> ::= <atomic_node> <multitask_node></code>
<code><node_id_clause> ::= node_id <natural_number></code>
<code><node_identifier_clause> ::= [<identifier_clause>] [<short_identifier_clause>]</code>
<code><permanent_reaction_statement> ::= permanent_reaction "(" <guard_clause> <node> ")"</code>
<code><reaction_node> ::= <basic_atomic_node> <basic_multitask_node></code>
<code><regular_transition_statement> ::= regular_transition "(" <node> ")"</code>
<code><short_identifier_clause> ::= short_identifier <identifier></code>

<code><temporary_reaction_statement> ::= temporary_reaction "(" <guard_clause> <reaction_node> ")"</code>
<code><unreachable_node> ::= unreachable "(" (<basic_multitask_node> <basic_atomic_node>) [<unreachable_reference_clause>] {<unreachable_reaction_clause>} ")"</code>
<code><unreachable_reaction_clause> ::= (conditional_transition permanent_reaction temporary_reaction) "(" <guard_clause> <node_id_clause> ")"</code>
<code><unreachable_reference_clause> ::= regular_transition "(" [multitask_node] <node_id_clause> ")"</code>

Appendix C

The following is the VSL file listing of the simple abstract scenario described in Section 5.3.

```
scenario name "simple_scenario"
(
  entity
  (
    entity_id 2
    transient 203.0 10.0
    identifier "US Cmd"
    entity_type "DistFOR"
    entity
    (
      entity_id 6
      transient 21.0 75.0
      identifier "DefendRWA"
      entity_type "AH64A"
      behavior
      (
        start_node
        (
          node_id 7
          transient 15.0 15.0
          conditional_transition
          (
            guard "OnOrder Command"
            node_id 8
            transient 95.0 125.0
            identifier "FlyRoute"
            conditional_transition
            (
              guard "Duration(75);Unit('sec')"
              node_id 9
              transient 248.0 149.0
              identifier "Hover"
              conditional_transition
              (
                guard "Duration(60);Unit('sec')"
                node_id 10
                transient 401.0 169.0
                identifier "Land"
              )
            )
          )
        )
      )
    )
  )
  entity
  (
    entity_id 5
    transient 20.0 30.0
```

```

    identifier "DefendFWA"
    entity_type "F14D"
    behavior
    (
        start_node
        (
            node_id 6
            transient 15.0 15.0
            conditional_transition
            (
                guard "OnOrder Command"
                node_id 11
                transient 121.99999999999999 109.0
                identifier "Ingress"
                conditional_transition
                (
                    guard "Duration(75);Unit('sec')"
                    node_id 12
                    transient 293.0 164.0
                    identifier "Return to Base"
                    short_identifier "Return"
                )
            )
        )
    )
)
entity
(
    entity_id 1
    transient 10.0 10.0
    identifier "USSR Cmd"
    entity_type "OPFOR"
    entity
    (
        entity_id 3
        transient 20.0 30.0
        identifier "AttackFWA"
        entity_type "MIG29"
        behavior
        (
            start_node
            (
                node_id 4
                transient 15.0 15.0
                conditional_transition
                (
                    guard "OnOrder Command"
                    node_id 13
                    transient 145.0 122.0
                    identifier "Ingress"
                )
            )
        )
    )
)

```


Bibliography

1. Ashby, Michael R. *Tool-Based Integration and Code of Object Models*. MS thesis, AFIT/GE/ENG/00M-02. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000
2. Bartley, Carolyn R. *A Visual Language For Composable Simulation Scenarios*. MS thesis, AFIT/GCS/ENG/03-03. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2003
3. Biddle, Mark and Constance Perry. "An Architecture for Composable Interoperability," *Proceedings of the 2000 Spring Simulation Interoperability Workshop*, 2000
4. Breighner, Lawrence A. *A Semantic Interface to Scenario Component Reuse in DoD Simulation Systems*. MS thesis, AFIT/GCS/ENG/01M-01. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2001
5. Canli, Hakan. *A Visual Meta-Language for Generic Modeling*. MS thesis, AFIT/GCE/ENG/02M-1. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2002
6. Colonese, Emilia M. *Methodology for Integrating the Scenario Databases of Simulation Systems*. MS thesis, AFIT/GCS/ENG/99J-03. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1999
7. Fu, Daniel and Ryan Houlette. "Putting AI in Entertainment: An AI Authoring Tool for Simulation and Games," *IEEE Intelligent Systems*: 81-84 (July/August 2002)
8. Fu, Daniel, Ryan Houlette and Randy Jensen. "A Visual Environment for Rapid Behavior Definition," *Proceedings of the 2003 Conference on Behavior Representation in Modeling and Simulation*, 2003
9. Glinz, Martin, Stefan Berner and Stefan Joos. "Object-oriented modeling with ADORA," *Information Systems*, Vol 27: 425-444, the 13th international conference on advanced information systems engineering. Oxford, UK. Elsevier Science Ltd (September 2002).

10. Hill, R. R., J. O. Miller and G. A. McIntyre. "Applications of Discrete Event Simulation Modeling to Military Problems," *Proceedings of the 2001 Winter Simulation Conference*, 780-788, 2001
11. Koziarz, Walter A., Lee S. Krause and Lynn A. Lehman. *Automated Scenario Generation*, 2002
12. McDonald, Jeffrey T. *Agent Based Framework for Collaborative Engineering Model Development*. MS thesis, AFIT/GCS/ENG/00M-16. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000
13. Noe, Penelope A. *A Structured Approach to Software Tool Integration*. MS thesis, AFIT/GCS/ENG/99M-14. Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1999
14. Paige, R. F., J. S. Ostrof and P. J. Brooke. "Principles For Modeling Language Design," *Information and Software Technology*, Vol 42: 665-675, UK, July 2000
15. Raphael, Marc J. *Knowledge Base Support for Design and Synthesis of Multi-Agent Systems*. MS thesis, AFIT/GCS/ENG/00M-21. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000
16. Stottler Henke Associates, Inc. *SimBionic Manual*, San Mateo, CA (December 2003)
17. Smith, Roger D. *Essential Techniques for Military Modeling & Simulation*, 1998 Winter Simulation Conference (October 1998)
18. Stratton, Phillip. *A Metrics-based Analysis of Interface Usability Improvements by Applying Intelligent Agents*. MS thesis, AFIT/GCS/ENG/99M-18. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1999.
19. Webber, Robert. *A Methodology for Extracting a Common Object Model for Simulation Systems*. MS thesis, AFIT/GCS/ENG/99M-20. Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1999.

20. Xia, Yong. *A Language Definition Method for Visual Specification Languages*. Ph.D. dissertation, Institut für Informatik, Universität Zürich, Switzerland, February 2004.

Vita

Master Sergeant Daniel E. Swayne was born in March 1959 in Griffiss AFB, New York. He graduated from Mulvane High School in 1977 and received his Bachelor of Science Degree in Computer and Information Science in 1996 from Troy State University in Montgomery, Alabama.

He enlisted in the United States Air Force in 1984 and has served as a computer programming specialist in support of various systems including Processing and Classification of Enlistees (PACE) and the Weighted Airman Promotion System (WAPS).

In August 2002, he began work towards a Master of Science Degree in Computer Science at the Graduate School of Engineering and Management, Air Force Institute of Technology at Wright-Patterson AFB, Ohio. He is a member of Alpha Sigma Lambda and Eta Kappa Nu honor societies.

Upon graduation he will be assigned to the Standard Systems Group at Maxwell AFB-Gunter Annex, Alabama.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 074-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) 14-09-2004		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From – To) June 2003 – August 2004	
4. TITLE AND SUBTITLE VISUAL UNIFIED MODELING LANGUAGE FOR THE COMPOSITION OF SCENARIOS IN MODELING AND SIMULATION SYSTEMS				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Swayne, Daniel E., Master Sergeant, USAF				5d. PROJECT NUMBER 2003-070	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way, Building 641 WPAFB OH 45433-8865				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCE/ENG/04-19	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Jack Sheehan Defense Modeling and Simulation Office 1901 North Beauregard Street, Suite 500 Alexandria, VA 22311-1705 (703) 998-0660x448 jsheehan@dmsso.mil				10. SPONSOR/MONITOR'S ACRONYM(S) DMSO	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT The Department of Defense uses modeling and simulation systems in many various roles, from research and training to modeling likely outcomes of command decisions. Simulation systems have been increasing in complexity with the increased capability of low-cost computer systems to support these DOD requirements. The demand for scenarios is also increasing, but the complexity of the simulation systems has caused a bottleneck in scenario development due to the limited number of individuals with knowledge of the arcane simulator languages in which these scenarios are written. This research combines the results of previous AFIT efforts in visual modeling languages to create a language that unifies description of entities within a scenario with its behavior using a visual tool that was developed in the course of this research. The resulting language has a grammar and syntax that can be parsed from the visual representation of the scenario. The language is designed so that scenarios can be described in a generic manner, not tied to a specific simulation system, allowing the future development of modules to translate the generic scenario into simulation system specific scenarios.					
15. SUBJECT TERMS Simulation, Combat Simulation, Simulation Languages, Models					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			MICHAEL L. TALBERT, Lt Col, USAF
U	U	U	UU	110	19b. TELEPHONE NUMBER (Include area code) (937) 255-6565, ext 4716 (Michael.Talbert@afit.edu)

