USING ONTOLOGIES TO SUPPORT INTEROPERABILITY IN FEDERATED SIMULATION

A Thesis
Presented to
The Academic Faculty

By

Tarun Rathnam

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Mechanical Engineering

Georgia Institute of Technology
August 2004

USING ONTOLOGIES TO SUPPORT INTEROPERABILITY IN FEDERATED SIMULATION

APPROVED:

Dr. Christiaan J.J. Paredis (Chair)

Assistant Professor,
Mechanical Engineering

Dr. Bert A. Bras

Professor,
Mechanical Engineering

Dr. Richard M. Fujimoto

Professor,
College of Computing

Dr. Russell S. Peak

Senior Research Scientist,
Manufacturing and Research Center

DATE APPROVED: 18TH AUGUST 2004

DEDICATION

To my parents—for their unending love, support and encouragement.

ACKNOWLEDGEMENT

In my life thus far, I have been faced with many a challenge—it seems as if every subsequent challenge I have taken on has been ‘he hardest one yet’. The success I have enjoyed in overcoming any and all challenges is on account of the support of many that I hold in high regard. The same applies to my completing this thesis. I use this opportunity to express my gratitude towards those that have had a significant hand in my completing this body of research, and more importantly, my development as a person.

I am forever indebted to my advisor and mentor, Dr. Chris Paredis, for having given me the opportunity to further myself immensely. Chris, to me you are a beacon of light and a pillar of support; I have garnered so much from your counsel and reached what I thought to be ‘unreachable’ goals with your encouragement and faith.

I extend my thanks to my committee members, Dr. Bert Bras, Dr. Richard Fujimoto and Dr. Russell Peak for their valuable feedback with regard to this thesis. Also, I owe thanks to Manas Bajaj for being a critical eye and combing through the algorithms in this thesis. This work has been supported by Sandia National Laboratories, whom I thank for their input and support. Specifically, I would like to acknowledge Daniel Fellig—though we haven’t met in person thus far; our long and frequent conversations over the phone have significantly contributed to improving my thesis.

Over the past two years, I have had the honor of working with a wonderful family of peers at the SRL, whom I thank collectively for their mentorship and support. I walk away from the SRL ‘nest’ having learned two invaluable lessons—(i) there is still much to learn, and (ii) never settle to be mediocre.

A special thank you to Rich Malak and Steve Rekuc—its been an interesting adventure since we all arrived here at Georgia Tech, and my completing this adventure is largely due to your support. To my close friends Chris Williams, Matt Chamberlain, Marco Fernandez and Scott Duncan, thank you for keeping me sane over the last couple of years and for improving my gaming skills, but most importantly, for your friendship and support. For giving me the ‘prodding’ that I occasionally require, I offer my heartfelt thanks to Benay Sager, Scott Cowan and Kannan Sockalingam.

Finally, I owe the greatest thanks to my family. As I type this acknowledgement, I realize that it because of their encouragement, love and the ideals they have bestowed upon me that I have made it thus far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

Aggregate Data type	Used to instantiate n:1 relationships between the attributes of simulation objects and events. The data type is an aggregation of the n attributes that need to be simultaneously mapped.
AFF	Agile FOM Framework; a framework to support the simplified reuse of federate simulation models in multiple HLA federations. In this framework, converters are instantiated to transform between FOM and SOM representations of exchanged information.
ATC	Air Traffic Controller; a system that communicates with aircraft in the local airspace of an aircraft, so as to manage the safe and efficient departure, arrival and passing of multiple aircraft. This system is modeled as a federate simulation in the Air Traffic federated simulation.
Air Traffic Federated Simulation	The system-level simulation of an airport being designed, so as to observe the emergent behavior of air traffic control & ground traffic control and ground service sub-systems, together, in response to different volumes of air traffic.
Attribute	A property of one or more simulation object or event. Each attribute contributes to the description of an object or event, and is modeled in an ontology as a slot.
Big-O Notation	The standard fashion to express the theoretical complexity of an algorithm.
BOM Framework	Base Object Model Framework; a framework that uses a piece-part approach to support reuse in the development of HLA FOMs.
Common Information Model	An information model that specifies the common representation of all shared entities in a federated simulation. This model is part of a FONT.
Cost of a Vertex	Used as a measure of the extent of information loss in transformations between federate attributes through a given common representation.

Dijkstra's Algorithm	Determines the shortest path between two vertices in a connected graph. This algorithm is employed to help define the common information model in a FONT.
Directed Graph	A graph whose edges are unidirectional.
DoD	Domain of Discourse; a bounded set of concepts within the universe of discourse.
Edge	A connection or relationship between two vertices in a graph. An edge may be uni/bi directional.
Event	An occurrence during the execution of a simulation. Events are non-persistent; they are of consequence only for a single time stamp of the simulation clock.
Federate	A single simulation model that interoperates with other simulations in a federation.
Federated Simulation	The parallel execution of a group of simulations wherein information is exchanged between individual simulations at run-time.
Federation	A collection of interoperating federate simulations
FEDEP Model	Federation Development Process Model; the systems engineering process model developed by the DoD for building HLA federations.
FOM	Federation Object Model; an information model that describes the set of objects, interactions and attributes that are shared across a federation.
FONT	Federation Ontology; a semantically rich information model that describes the federate and common representations of shared simulation entities in a federation, and the relationship between these representations.
Forest	A collection of connected sub-graphs.
Frame	The generic information structure used to model concepts in an ontology. Frames can be specialized into metaclasses, classes and instances.

Frame-based Knowledge Model	A knowledge representation model widely used in ontology specification. The principal elements of this model are the Frame and the Slot.
Graph Algorithms	Algorithms to solve problems relevant to graph-based representations and graph theory.
Graph Theory	The field of mathematics that deals with the use of diagrams or graphs to study the arrangement of objects and the relationships between them.
GRIT Algorithm	The graph based algorithm that uses knowledge contained within a FONT to generate a common information model and SONT-Common relationships.
GTC	Ground Traffic Controller; a system that manages the safe and efficient movement of multiple aircraft between runways and gates. This system is modeled as a federate simulation in the Air Traffic federated simulation.
HLA	High Level Architecture; a framework for federated simulation developed by the United States Department of Defense.
Lossiness	A transformation from one simulation entity to another that involves loss of information is a lossy transformation.
Mapping	The knowledge required to convert an instance of one entity to that of another is captured in a mapping between them.
Matching	The knowledge as to which two simulation entities equate to, or are related to each other is captured as a match between them.
Metaclass	A template for specifying classes. A metaclass is an entity whose instances are classes.
Metamodel	A model of a model. Metamodels define a vocabulary for expressing models.
Metaslot	A template for specifying slots. A metaslot is an entity whose instance are slots.

Object	A persistent entity modeled though the entire length of a simulation execution.
OMT	Object Model Template; a specification for the instantiation of HLA FOMs and SOMs, adopted as an IEEE standard.
Ontology	A formal, explicit specification of a conceptualization, which refers to the concepts in a given domain of discourse and the relationships between them.
Publish & Subscribe	These are services provided by an RTI for exchanging information between federate during the execution of a federated simulation.
Representational Compatibility	The task of relating between disparate representations of shared concepts in a federation, so that information can be exchanged in a consistent manner at run-time.
RTI	Run Time Infrastructure; an operating environment for distributed federated simulations, which provides services for exchanging information between federate simulations.
Semantics	The study of meaning. Semantics is opposed to syntax, in that the former pertains to what something means, while the latter pertains to the structure in which something is expressed.
Simulation Model	A computerized, mathematical or programmatic model of a real world system, used to predict the behavior of that system in a certain environment.
Slot	The information structure used to capture relationships between concepts (frames) in the frame-based architecture. All classes in an ontology are described in terms of their slots.
SOM	Simulation Object Model; an information model in which the set of objects, interactions and attributes in a given HLA federate simulation domain are documented.
SONT	Federation Ontology; a semantically rich information model that describes the federate and common representations of shared simulation entities in a federation, and the relationship between these

	representations.
SONT-Common Relationship	A relationship between a SONT entity (an attribute, object or event) and its corresponding common representation. These relationships are automatically generated in the execution of the GRIT algorithm.
SONT-SONT Relationship	A relationship between two SONT entities (attribute, object, event or data type), which is specified explicitly by the federation developer.
Subsumption	An inheritance relationship between concepts, where a more specific concept is incorporated under a more general category. In the context of an ontology, class B subsumes class A if the set of slots in the domain of class B includes all slots in the domain of class A.
Technical Interoperability	A condition in which federate simulations can exchange information with each other in a consistent manner, at runtime. The key aspects of technical interoperability are representational compatibility, synchronization and interaction with the RTI.
Transformation stub	A procedural relationship to convert information between disparate representations.
Validation Square	A research validation process that is anchored in the relativistic, holistic school of epistemology, where scientific knowledge is defined as socially justifiable belief, and knowledge validation is postulated to be a process of building confidence in its usefulness with respect to a purpose.
Vertex	The central concept in a graph, a vertex represents an object, and may be the end point of one or more edges.
World Ontology	The metamodel for the specification of SONTs and FONTs. The World Ontology specifies a vocabulary for describing a given simulation domain.
XML	Extensible Markup Language; a flexible language that can be used to create standard information formats and share both the format and the data on the World Wide Web. XML is developed by the World Wide Web Consortium (W3C).

SUMMARY

A vast array of computer-based simulation tools are used to support engineering design and analysis activities. Several such activities call for the simulation of various coupled sub-systems in parallel, typically to study the emergent behavior of large, complex systems. Most sub-systems have their own simulation models associated with them, which need to interoperate with each other in a federated fashion in order to simulate system-level behavior. The run-time exchange of information between federate simulations requires a common information model that defines the (representation of) entities (simulation objects and events) that simulators can publish or subscribe to. However, most federate simulations employ disparate representations of shared concepts. To address the problem of disparate representations, federate simulation developers must agree upon a common representation for concepts that are exchanged at runtime and modify their simulation models accordingly. Furthermore, it is often necessary, especially for legacy simulators, to implement transformation stubs that convert objects and events from the common representation to those used in the legacy implementation. The tasks of defining a common representation for shared simulation concepts, modifying individual simulations and building translation stubs around them can add significant time and cost to defining a system-level simulation.

In this thesis, a framework to support automation in the process of achieving interoperability between federate simulations is developed. This framework uses ontologies to capture knowledge about the semantics of different simulation concepts in a

formal, reusable fashion. Using these semantics, a common representation for shared simulation entities, and a corresponding set of transformation stubs to convert entities from their federate to common representations (and vice-versa) are derived automatically. In capturing the description of simulation models and the relationships between them in a formal manner, this framework also supports the simplified re-use of federate simulations in multiple federations. As a foundation to this framework, a schema to enable the capture of simulation concepts in an ontology is specified. Further, steps are elaborated for capturing knowledge as to the relationship between different federate simulation entities. Finally, a graph-based algorithm is developed to extract the appropriate common information model and transformation procedures between federate and common simulation entities.

As a proof of concept, this framework is applied to support the development of a federated air traffic simulation. To progress with the design of an airport, the combined operation of its individual systems (air traffic control, ground traffic control, and ground-based aircraft services) in handling varying volumes of aircraft traffic is to be studied. To do so, the individual simulation models corresponding to the different sub-systems of the airport needs to be federated. The ontology-based framework is employed to support the development of this federation.

CHAPTER 1

INTRODUCTION

Computer-based simulation is pervasive in the systems realization process, and provides an effective means of predicting system behavior that is a favorable alternative over physical experimentation. In the development of complex systems, distributed and federated simulation is instrumental in supporting the development and integration of sub-systems in a time and cost effective manner. In this chapter, we highlight the importance of federated simulation in systems engineering (Section 1.1), and discuss key issues and challenges faced in developing federated simulations. We motivate the research presented in this thesis by elaborating challenges faced in achieving interoperability between federate simulations, specifically concerning the fact that simulations employ disparate representations of coupled concepts. Achieving interoperability between simulations can be an effort-intensive task that is in the critical path of a system's development. In this context, we pose research questions and accompanying hypotheses so as to investigate how the challenges associated with attaining interoperability in simulation federations can be alleviated (Section 1.2). A plan for validating the proposed hypotheses is laid out in Section 1.3. Finally, the layout of the remainder of this thesis, in the context of executing the validation plan, is presented in Section 1.4.

1.1 The Importance of Simulation in Design

The design of complex engineering systems is a multi-step process in which a set of design goals and requirements are transformed into a functional system, whose behavior in its intended environment meets the above mentioned goals. Several methodologies to guide the design of complex systems have been developed, such as the systematic design method by Pahl and Beitz (1996) and the systems engineering approach (Forsberg and Mooz 1992). One common activity in these methodologies is *modeling and simulation*. In the words of Bernard Zeigler, "Modeling refers to the process of organizing knowledge about a given system" (Zeigler 1990). Simulation is the process of performing experiments on a model. By performing simulations, knowledge about a system is gained. Therefore, "it can be said that modeling and simulation are the most central activities that unite all scientific and engineering endeavors" (Cellier 1991). In the age of electronics, computer-based modeling and simulation have become increasingly popular and ubiquitous in engineering design. When analytical techniques and physical experimentation are not viable, simulation provides an avenue by which the behavior of a system can be studied. The key merits of simulation in the context of systems design are listed as follows:

- **To explore and focus the solution space corresponding to a given design problem, at a reasonable cost:** Systems design can be viewed as a process in which the solution space for a given design problem is progressively constrained until a point solution is obtained. In order for a designer to make educated

decisions about a given aspect of a system, it is important to explore the solution alternatives available. Based on this exploration, the designer can then pick the best suited alternative, and focus all subsequent design activities on further refining the solution space. The exploration of a large solution space can be a very time and cost intensive operation. For example, the time and workforce required to develop physical models and conduct experiments on them can be tremendous, especially for expensive, complex systems such as aircrafts and satellites. In the design of business processes and services, it is usually economically infeasible to set up different pilot services and determine which works best. Computer-based simulation can be used to perform such explorations at a lower cost (in terms of time, workforce and monetary value). At the early stages of design, intended behavior and function simulation can be applied to gauge how different concepts may be used to address functional requirements of a system at a relatively high level of abstraction. As design progresses, designers can simulate the form of the system, optimize its attributes at a fine level of granularity and verify that its expected behavior is in line with how it was intended to behave.

- **To reduce uncertainty about the system and reduce the changes of design failure:** The design process can be viewed as information and knowledge driven—as the design of a system progresses, designers apply knowledge to add and transform information about the system until a complete, detailed specification is achieved. The growth of explicit information and knowledge about a system leads to a reduction in *epistemic* uncertainty (uncertainty

characterized by lack of knowledge, as opposed to inherent randomness). Simulation is basically an activity in which knowledge (models) are applied to existing information about a system and its environment, to obtain new information and knowledge about that system (its behavior). Therefore, simulation can be used to reduce epistemic uncertainty associated with the different aspects of the system (its components, its behavior, its form etc.) (Aughenbaugh and Paredis 2004).

- **To predict and analyze a system's behavior in an artificial environment:** Often times, the environment in which a system is to operate and interact with is not fully known to the designer. Several systems create or change their environment as they operate. Other systems, once they are built, must interact with humans or other existing artificial or natural systems. Through simulation, the behavior a system being designed and its interactions with its environment can be predicted. Simulation is an effective tool to aid in studying a system's sensitivity to varying environmental stimuli, so as to design an end-product that is robust to noise.

- **To guide the development of multiple sub-systems so that the overall system behavior meets the design goals:** As engineered systems become increasingly large-scale and complex, they begin to span several engineering domains and cannot be designed by an individual. For the development of such systems, a holistic, hierarchical decomposition approach, namely systems engineering, is

taken on. Systems engineering models, such as the ‘Vee’ model (Forsberg and Mooz 1992) (illustrated in Figure 1.1), prescribe the decomposition of a complex system into a hierarchy of sub-systems, each of which may be coupled with other sub-systems. A design-to specification for each sub-system is developed and handed off to individual development teams that are experts in their own domains. At this point, an important challenge that arises is in making sure that the design of individual sub-systems progress in such a fashion that when integrated, the overall system meets its behavior objectives. We have already established that simulation is instrumental in progressing through the development of each sub-system. In addition, simulation has a large hand to play in coordinating the concurrent design of multiple sub-systems. Very often, there is significant coupling between the various sub-systems of a complex system. That is, the decisions made regarding the attributes of one sub-system impact the design of other sub-systems. There may be many cost and performance trade-offs that should be investigated as they play an important role in determining the behavior of the overall system. Therefore, it is critical to make decisions about coupled design attributes in a concurrent fashion.

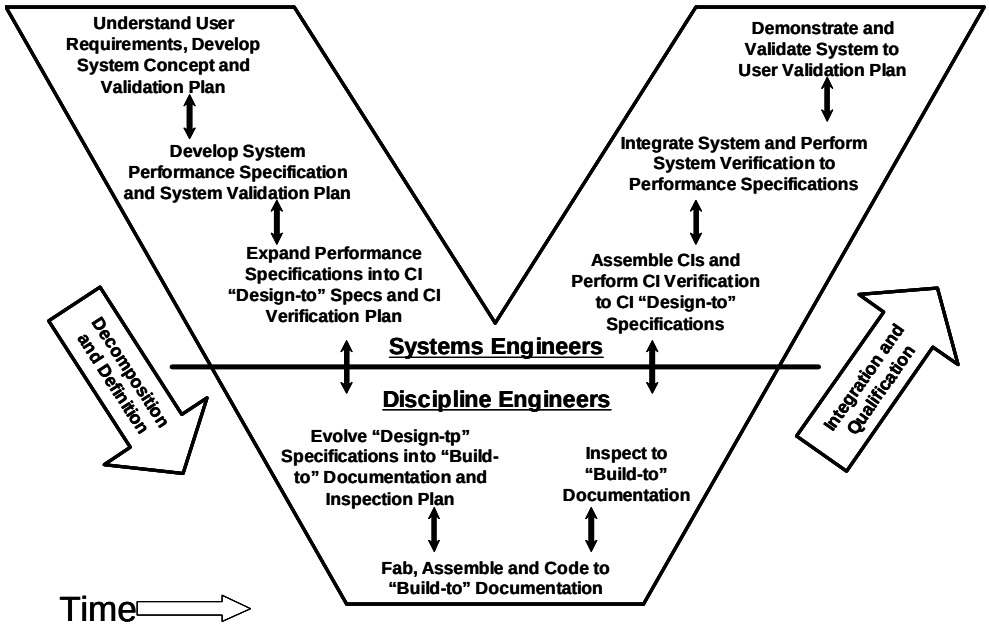


Figure 1.1: The 'Vee' Model for Systems Engineering

1.1.1 Distributed and Federated Simulations

To predict system-level behavior when exploring the solution space for coupled sub-systems, *distributed* and *federated* simulation systems are often useful. A distributed simulation is simply one that is executed on multiple computers that are geographically distributed. A federated simulation is a system-level virtual experiment in which multiple sub-system or *federate* simulation models participate. The idea of a federated simulation is akin to that of political federations, which are conglomerates of regional governments with a common central government. Regional governments enforce policies in their individual districts and work with each other to enforce state-wide policies. Similarly, in a federated, distributed simulation, individual models (developed by disparate, geographically distributed teams on their own computers) can be linked

together so as to be able to exchange information with each other when executed. A simulation model is referred to as a federate if information about the phenomena it models is ‘published’ to other simulations during their parallel execution. Similarly, a simulation is a federate if certain phenomena modeled in other (federate) simulations affect its execution. A collection of federate simulations that exchange information with each other comprise a federation. The concurrent execution of federate simulations such that they exchange information with each other is referred to as a federated simulation experiment.

Typically, the individual simulations in a federated simulation are discrete event simulations. The changes of state in a discrete event simulation are viewed to occur at discrete points in time. Interaction in a federated simulation is event-driven—the changes in state in one simulation are communicated in real-time to other coupled simulations, whose states can subsequently change. This interaction is message-based, where one simulation ‘publishes’ information in a message, which another federate can ‘subscribe’ to by receiving a corresponding message. In this manner, the interaction between coupled sub-systems and the resultant emergent behavior at the system-level is studied using federated simulations.

Federated simulation plays a key role in the systems realization process. We have already seen that such simulations can be used to study the behavior of complex systems, and support decision-making when sub-systems are involved. In doing so, federated simulations uncover unanticipated or emergent behavior that designers have no prior

knowledge of. Furthermore, these simulations do this at a reasonable cost; the only other way to study system-level behavior would be to create and integrate physical models, which is probably not feasible for most complex systems. Federated, distributed simulation also helps to facilitate system-level optimization, wherein the effect of tweaking sub-system level parameters on the overall system behavior is gauged. Similarly the reliability and robustness of a system and issues related to the integration of sub-systems developed in a distributed manner can be studied with the help of federated simulations.

As one might imagine, distributed simulation is used to support system realization in several application areas. For example, in the military community, federated simulation systems are used to conduct war gaming simulations to evaluate attack and defense strategies and to develop training environments for military personnel, wherein humans are part of a simulation federation. In the semiconductor design domain, distributed simulations are used to simulate gate-level logic interaction between components in large integrated circuits. But perhaps the quintessential example to highlight the use of distributed simulation is in the design of an airport. By any standard, a airport is a complex system, composed of several interacting sub-systems such as the air traffic control system, runway and taxiway systems, passenger service systems and communications systems. Following the systems engineering approach, the airport's specification is decomposed into specifications for each sub-system. Here, distributed simulation systems can be used to ensure that the subsystem level (executable) specifications (Aughenbaugh and Paredis 2004) together reflect those of the overall

airport being designed. While the individual sub-systems can be designed by disparate teams, they cannot be designed completely autonomously. Distributed simulation is used to study how these systems interact to perform the system-level operational requirements of the satellite. Since safety is a big issue with respect to airports, distributed simulation can be used to determine how the different systems behave in reaction to security breaches, emergencies, mishaps and so on. Note that the cost of performing such experiments physically would be tremendous. The airport example illustrates what is true for all complex systems developed following systems engineering models such as the 'Vee': Distributed, federated simulation plays a vital role in the systems development process.

The idea behind the federated approach to performing distributed, parallel simulations is to organize discrete event simulations such that they are reusable, by defining a message-based interface between them. Theoretically, this is a very efficient way to integrate a collection of simulation models and facilitate interplay between them. However, in practice, federate simulations are not readily reusable, as a result of which the development of federations is quite complex and requires significant effort. Significant complexity arises from the fact that federate simulations model coupled concepts in different ways. In the next section, challenges specific to developing simulation federations and facilitating communication between individual federate simulations are discussed as motivation for the research questions posed in this thesis.

1.1.2 Requirements of Federated Simulations

In the previous section, the significance of distributed and federated simulation to support the development of complex systems has been highlighted. Simulation federations facilitate the composition of and the interaction between sub-system level simulation models, so as to simulate the emergent behavior of the entire system. However, the task of developing federated simulations is not trivial. There are several issues that make distributed simulation challenging, such as (i) integrating component simulation models, and (ii) providing a means for real-time communication. These challenges are elaborated in this section. Specifically, the reader's attention is focused on the issue of developing and using simulation information models to support the integration of federate simulation models.

A key challenge in developing and executing distributed simulations is achieving *technical interoperability* between federate simulations. Technical interoperability refers to the capability of federate simulations to connect and exchange information with each other at run-time (Dahmann, Salisbury, Turrell et al. 1999). The chief elements of achieving technical interoperability are:

- Representational compatibility
- Run-time information exchange
- Time management coordination
- Security issues

As is explained in the paragraphs below, achieving interoperability requires significant effort, and involves the creation and application of compatible interfaces, middleware and information models.

Representational Compatibility: "Most simulation software systems live in isolation" (Cellier 1991). That is, disparate simulation environments exist in which domain-specific simulation models can be generated and executed. For example, CAD software tools such as ProEngineer and Catia facilitate geometric modeling and simulation for mechanical systems, while integrated circuit simulation is carried out on tools such as Cadence. Each of these domain-specific simulation environments employ disparate ways in which models are represented. Each software system may even define its own language for capturing models (example: VHDL and PSPICE for integrated circuit modeling). The fact that different simulation systems employ different representations is a major issue in linking coupled concepts between federate simulation models in a distributed simulation. Aside from the representational constraints enforced by the simulation system, simulation model developers themselves can represent concepts in multiple ways within their individual sub-system models. For example, the models corresponding to one sub-system may employ SI units of measurement, while that of another coupled sub-system may be expressed in the British Foot-Pound unit system.

The run-time exchange of information between distributed simulations requires a common information model that defines the (representation of) objects and events that simulators can publish or subscribe to (Morse 1996). To address the problem of disparate

representations, simulation developers create information models to document how concepts have been represented in their simulation models. Using these simulation models, the set of objects and events that are related and the discrepancies in their representations can be identified. Having identified these inconsistencies, model developers must agree upon a common representation for related concepts and modify their simulation models accordingly. Furthermore, it is often necessary, especially for legacy simulators, to implement translation routines (stubs) that convert the object and event types in this common information model to the object and event types used in the legacy implementation. The tasks of defining simulation information models, comparing them, coming to an agreement on how coupled or related concepts should be modeled, modifying individual simulations and building translation stubs around them adds to the cost of facilitating distributed simulation. As the complexity of the coupling between sub-systems increases, the time and effort invested into these tasks increases (Ryde and Taylor 2003). Given the competitive nature of the marketplace for complex products and services, being cost-effective and first-to-market are important goals of a distributed design team. Therefore, the task of integrating sub-system simulations in a cost and time effective manner becomes an important challenge in employing federated simulation to support the design of complex systems.

Run-Time Information Exchange: Aside from resolving representational inconsistencies across federate simulations, there is still the issue of facilitating communication between these simulations as they execute. When the state of a coupled entity in one simulation changes, that change should be reflected in other simulations,

whose states may change as a result. Therefore, a run-time infrastructure (RTI) is required to provide services in which the changes or updates of coupled objects and events are recognized and reported in real-time. Also, middleware has to be developed so that each federation simulation environment (which is responsible for executing a federate simulation model) can interface with the run-time infrastructure services, to indicate the changed state of a coupled simulation object or event. This middleware is also responsible for ensuring that information about coupled concepts is presented in an exchangeable representation. Developing an RTI and middleware is not a trivial task. Again, cost, time and effort is expensed to do so, which can be quite significant when the simulations are tightly coupled.

Time Management and Synchronization: Time management is a key element of technical interoperability in a federation. Time management deals with mechanisms that control the transient advancement of each federate simulation. The mechanisms must be in place so that information is conveyed between different simulations in a timely manner. Not all simulation execute at wall clock speed; when two coupled simulations advance at different rates, the true emergent system behavior is not simulated unless the cause and effect interactions between these coupled simulations are presented in the correct order. Therefore, individual simulation time advances need to be paced and coordinated (Dahmann, Fujimoto and Weatherly 1997). The solution to this problem is to control logical time-advances in a distributed simulation within the RTI. Each individual simulator (on which federate models are being executed) must then request time advances from the RTI. Time management issues add more overhead to the process of setting up a

distributed simulation. Along the same lines as time management is the issue of synchronization. Consider a distributed simulation consisting of three coupled sub-system level models. If Model A receives messages indicating a change in state of coupled simulation parameters from both Models B and C, one would need to determine which message to process first (Ryde and Taylor 2003). To do so, some mechanisms to order and synchronize message-passing between federate simulations must be developed. In the absence of such mechanisms, a given distributed simulation execution could end up in deadlock (Fujimoto 2000).

From the key challenges in achieving technical interoperability between federate simulation models, it is clear that developing and executing distributed simulations is not trivial. As the complexity of coupling between sub-system models increases, so does the difficulty of setting up a system-level simulation federation. That being said, it would be significantly beneficial if one or more of the tasks associated with achieving interoperability could be avoided when a federate simulation model is reused in another distributed simulation. This is a challenge in itself.

Having visited the various challenges associated with attaining technical interoperability, a context and motivation for the research presented in this thesis is laid out. Now the reader's focus is shifted to the research goals in this thesis—to address and mitigate a subset of the challenges associated with achieving interoperability between federates in distributed simulations. In the following section, the research questions (and hypotheses) posed and answered in this thesis are elaborated.

1.2 Research Focus and Questions

In the previous sections, the context (distributed simulation), and motivation (challenges in developing federated simulations) have been outlined. The purpose of this section is to clearly define the research contribution presented in this thesis. Having identified the key challenges in distributed simulation, research questions are posed with the intent of addressing (a subset of) those challenges. Hypotheses corresponding to the research questions are proposed and the overall vision for addressing specific challenges associated with federation development is presented.

It has been established that although distributed simulation is very powerful in supporting the realization of complex systems in a cost and time effective manner, the task of developing and executing distributed simulations itself can be quite cost and time intensive. Associated with the elements of technical interoperability is the overhead of tailoring simulation models and simulators to participate in a federated simulation. Mitigating the cost, effort and time required to conduct these tasks would significantly increase the efficacy of distributed simulation in supporting complex systems design. This is essentially the motivation behind the research presented in this thesis. However, addressing all the challenges associated with achieving technical interoperability is a gargantuan problem. Therefore, this research is focused on addressing a sub-set of these challenges. In this thesis, the research conducted is focused on addressing the issue of representational compatibility between federate simulations. Specifically, we investigate how it may be possible to reduce the cost, time and effort required to achieve consistent

information transfer between simulations employing different representations of related concepts.

One way to reduce the overhead incurred in performing the tasks associated with achieving representational compatibility would be to automate them. These tasks include developing information models corresponding to individual simulation models, identifying relationships between concepts, identifying discrepancies between the representations of related concepts and rectifying all representational inconsistencies. At the outset, it should be noted that these tasks cannot be completely automated. In order to develop an information model for a federate, knowledge of the concepts modeled in that federate is required. This knowledge cannot be generated by a computer; it must be provided by humans, probably those that developed the federate simulation models. Furthermore, a computer does not have prior knowledge of which concepts in federate simulations are meant to be related (effectively which aspects of sub-systems are coupled). While a computer cannot completely automate the process of achieving representational compatibility between federate simulations, it can conceivably support this process. The use of a system to partially automate this overall process would significantly mitigate the tedium, time and hence cost associated with ensuring representational compatibility in a distributed simulation. In this thesis, the realization of such a system is investigated. Hence, the primary research question to be answered is as follows:

Question 1: How and to what extent can the process of achieving representational compatibility between simulations in a federation be automated?

It has already been established that knowledge is required to perform tasks associated with attaining representational consistency. While a computer cannot create knowledge on its own, it can apply knowledge provided by a human being. Therefore, an important task in the above stated automation problem is the capture of knowledge such that it can be used by a computer. The development of knowledge-based systems to capture human-provided knowledge such that it is interpretable by a computer is a fast-growing research field. As systems engineering becomes increasingly knowledge-intensive and collaborative, the need for computational frameworks to enable engineering product development, by effectively supporting the formal representation, capture, retrieval and reuse of product knowledge, has become critical (Szykman, Sriram and Regli 2001). Several research efforts have been taken on to address the capture and use of knowledge related to systems design. Specifically, (Horvath and Van Der Vegte 2003) and (Liang and Paredis 2004) have applied the use of semantic technologies to formally capture design related metadata. Semantic technologies refer to languages and models to capture metadata and tools to apply them. These technologies were pioneered by the World Wide Web community in the effort to represent knowledge about web content in a machine-processable form (Davies, Fensel and Van Harmelen 2003). The goal of semantic technologies is to allow different agents (software or human agents) to interoperate and share meaning. Just as semantic technologies have been leveraged to capture knowledge related to the design of a system, they can conceivably be applied to capture knowledge relating to the design of a federated simulation.

A key constituent of semantic technologies are Ontologies. Basically, Ontologies are information models developed to capture metadata about web content. An ontology is defined as a specification of a conceptualization—it captures the different concepts and relationships between them in a given Domain of Discourse (DoD) (Gruber 1993). The key ingredients that make up an ontology are a vocabulary of basic terms, a precise specification of what those terms mean and how they relate to each other. By organizing knowledge in a discrete layer for use by information systems, ontologies enable communication between computer systems in a way that is independent of the individual system technologies, information architectures and applications (Berners-Lee, Hendler and Lassila 2001; TopQuadrant 2003).

In this thesis, the use of ontologies to capture knowledge requisite to support the process of achieving interoperability between simulations is proposed and demonstrated. Ontologies can be used as a medium for representing knowledge about the concepts defined in a federate simulation. Such ontologies would describe the semantics (or meaning) of concepts defined in each simulation model participating in a given federation. Given a formal definition of the meaning of each concept, the representational differences between two related concepts can be identified automatically. That is, software can be developed to query the knowledge captured in an ontology and determine facts about each concept in a simulation model. From these facts, the software can automatically determine if two concepts have equivalent representations.

Above and beyond this, the knowledge contained within simulation ontologies can be applied to determine the representational mapping between related concepts. That is, a transformation to convert information from one representation to another could possibly be derived in an automated fashion, based on the existing set of relationships captured in an ontology. The automated generation of such transformations is very helpful in mitigating the cost of achieving interoperability. Recall that in a distributed simulation, simulations models can have disparate representations of related concepts, and translation stubs must be built around them to convert information sent to (and received from) the RTI into a common form that ensures consistent information transfer when the distributed simulation is executed. In essence, these stubs can be developed automatically, to be employed by the middleware that interacts with the run-time platform. (A more detailed explanation as to how these transformations could be generated in an automated fashion is presented at a later point as the hypothesis to another research question).

The vision for an ontology-based framework that supports automation in achieving interoperability (representational compatibility to be specific) in a federated simulation is illustrated in Figure 1.2. The key components in this framework are (i) ontologies for participant simulation models (in which the objects and events defined in federate simulation model are captured), (ii) a federation-level ontology in which a common representation for shared simulation concepts is captured and (iii) a system to apply the semantics of individual simulation concepts to determine representational relationships between them. Federate simulation developers provide knowledge about the representation of simulation concepts in their models through the specification of

simulation models. When a federated simulation is to be created, federation developers can specify the set of related federate simulation concepts, based on which a common information model for consistent run-time information exchange is developed. The common representation for shared simulation concepts is captured in a federation-level ontology. Finally, the semantics captured in the federate and federation ontologies are used to automatically create transformation stubs to convert entities between their common and federate representations.

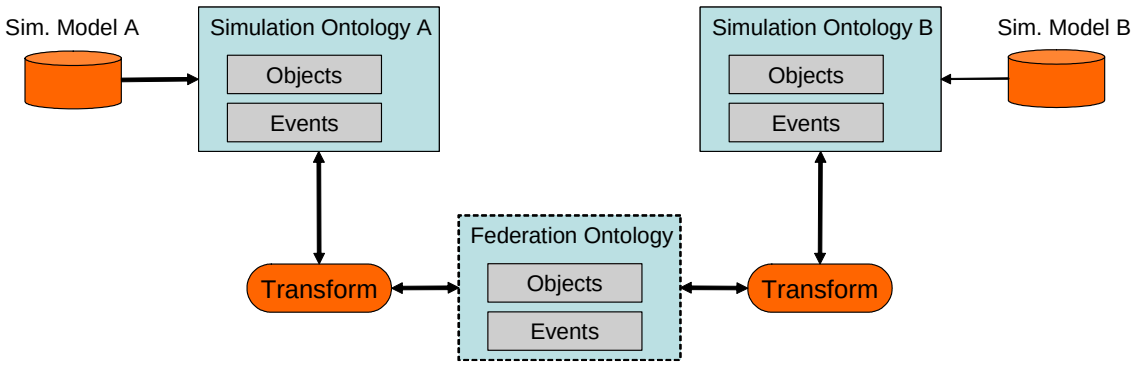


Figure 1.2: Vision for an Ontology-Based Framework to Support Automated Simulation Integration

We project that there are certain limitations as to the extent of automation that can be achieved using the ontology-based approach. Clearly, complete automation of the process of achieving representational compatibility is impossible. Only that knowledge which has been captured in a machine-interpretable, formal representation (an existing ontology) can be applied to support achieving interoperability in an automated fashion. Capturing the entire set of knowledge requires a complex knowledge model. In the framework

discussed above, we seek to arrive at a knowledge model that strikes a balance between expressiveness (the level of semantics that can be captured) and efficiency (the effort required to capture these semantics in an ontology). Beyond some level of semantic 'richness', the task of capturing and applying knowledge to support automation becomes significantly complicated, but the resultant payoff is not significant. In the framework illustrated above, we envision the use of ontologies to capture the knowledge required to generate a common information model and associated transformation stubs for a majority of coupled concept scenarios. Specifically, we aim to automate the generation of a common information model and transformation stubs for related federate simulation entities that model the *same* concept. For a scenario wherein simulation entities that refer to distinct yet related concepts are coupled, a richer set of semantics are required to (knowledge about the conceptual relationship between the two simulation entities) in order to generate transformation stubs to convert between the two entities. We do not intent to support automation in such cases, given their infrequency and the significant added complexity of capturing and applying additional semantics. However, it is important to ensure that the required transformation stubs for such scenarios can be specified manually in an intuitive fashion.

Based on the idea of using ontologies to capture knowledge, the hypothesis proposed in connection with research Question 1 is as follows:

Hypothesis 1: Ontologies can be used to formally describe the semantics of concepts in a federate simulation model. These semantics can then be applied to generate a required common information model and associated transformation stubs in a partially automated fashion.

The framework and hypotheses posed above allude to more research questions concerning specific elements of the overall approach. Below, those questions and corresponding hypotheses are elaborated. The division of the overall research question stated above into a set of sub-questions helps to frame the research problem better and ensure that the work undertaken explicitly addresses key issues in automating the process of achieving interoperability in a distributed simulation.

In the framework outlined above, it is proposed that ontologies be employed as a tool to capture knowledge implicitly known by a distributed simulation developer, in a formal, reusable format. Once captured, this knowledge can be applied to automate the process of attaining representational compatibility in a federation. However it is not clear how this comes about. There are two specific areas that need to be focused upon in hypothesis 1—(i) the way in which knowledge about simulation concepts is to be modeled in an ontology, so as to support the automated inferencing of relationships between disparate representations of shared concepts, and (ii) the process by which knowledge captured in a simulation ontology is applied to infer these relationships. Addressing these issues is critical to answering the overall research question posed in the context of the proposed hypothesis. Therefore, two subordinate research questions are identified as follows:

Question 2: How should simulation concepts be represented in an ontology to support achieving interoperability?

Question 3: How can the transformations between two representations of a simulation concept be derived in an automated fashion?

For the relationship between two entities to be inferred in an automated fashion, the semantics of those two entities should be unambiguous. In order to differentiate between the 'meaning' of two concepts, both those concept must be defined using the same vocabulary (at some level of abstraction). The same is true for determining the representational relationship between two entities in a simulation federation. Therefore, all concepts in a simulation model should be described using a common baseline vocabulary.

Ontologies capture knowledge about a DoD in discrete layers. A set of terms (and relationships) comprising all the concepts defined in one ontology can be used as a metamodel for specifying knowledge about individual entities in the DoD. In other words, an ontology defines a vocabulary to describe individual instances in a domain. At a lower level of abstraction, these instances can be used as a vocabulary to describe more concrete entities. For example, one ontology may define the semantics of a vehicle, which can be used as a metamodel for capturing information about sedans and coupes. These semantics could then be applied to describe the Mercedes SL55 coupe and the BMW M5 sedan, and differentiate between them based on the relationships defined (between sedans and coupes) at a higher level of abstraction.

This layered approach to capturing knowledge can be applied to capture the semantics of simulation concepts such that relationships between shared concepts can then be inferred automatically. A baseline ontology can be developed that serves as a metamodel for capturing all federate simulation domains. This metamodel should define the notion of simulation concepts (objects, events and their attributes) without limiting expressiveness (so that the definition of individual simulation concepts is not overly constrained). If all federate simulation concepts are defined in terms of this metamodel, the meaning of every concept in every simulation ontology is unambiguous, and the relationship between two coupled concepts in a federation can be derived in an automated fashion. Hence, the hypothesis corresponding to research question 2 is termed as follows:

Hypothesis 2: A metamodel for specifying simulation ontologies can be developed. The set of concepts and relationships between them defined in this metamodel form a vocabulary for describing simulation ontologies. If all simulation concepts are modeled using the same vocabulary, the relationships between two coupled simulation concepts in a federation can be inferred in an automated fashion.

Having discussed the capture of simulation concepts in an ontology, a hypothesis corresponding to research question 3 is elaborated below. As mentioned earlier, the relationship between two disparate representations of a concept in a distributed simulation can be derived in an automated fashion, based on the existing semantics. Relationships between federate simulation objects and events are to be specified in terms of a common information model to facilitate consistent exchange of information at run-time. That is, a common representation for all shared concepts is to be defined, and relationship between two federate simulation objects is to be captured as relationships

between each of them and their common representation. Further, transformation routines or stubs must be generated as procedural forms of these relationships.

All simulation entities are specified in terms of a common set of concepts (objects, events, attributes, and primitive data types) defined in the same metamodel. Within this metamodel, a set of relationships between these concepts are defined, including equivalence, inheritance, unit conversions, and other complex mathematical relationships. When a given simulation entity is modeled, an association relationship is instantiated between that entity and a set of concepts defined in the metamodel. Therefore, the relationship between two simulation entities can conceivably be derived as a chain of relationships— the association between those entities and the metamodel concepts plus the relationship between the specific metamodel concepts. In other words, existing relationships can be composed together to generate the required representational relationship between two simulation entities.

As mentioned earlier, in order to derive relationships between two simulation entities in a federation, a common representation of those entities is prerequisite. In the proposed ontology-based framework, a federation-level ontology can be developed in which a common representation for the set of coupled entities in a federated simulation is captured. A human must explicitly indicate which set of entities in the individual simulation ontologies are mapped to each other. A corresponding federation-level ontology comprising a non-redundant set of common object and event representations can then be developed. The instantiation of such a common information model can be

automated based on the notion that the common representation of a shared entity can be selected as one of its existing federate representations. The development of an algorithm to automatically determine a common representation for shared concepts in a federation is explored in this thesis.

Furthermore, software can be employed to apply the knowledge contained in the set of ontologies in a federation to generate transformations between the federate and common representations of simulation entities. An algorithm must be developed to determine relationships between federate entities and their common representation equivalents in an automated fashion. The application of graph theory and graph traversal algorithms to the development of such an algorithm is proposed. The approach envisioned is as follows: an algorithm can query ontologies associated with a given simulation federation to construct a connected graph that comprises the existing relationships. In this graph-based approach, simulation entities and metamodel concepts would be captured as nodes and the relationships between them as edges. Existing graph traversal algorithms could then be leveraged to find paths connecting those nodes between which relationships have to be derived (Kasyanov and Evstigneev 1994). In this manner, semantics can be exploited to infer relationships between simulation entities as a sequence or chain of existing relationships (Figure 1.3). Assuming that the relationships specified at the metamodel level are captured as procedures, the required transformation stubs can be generated automatically as well.

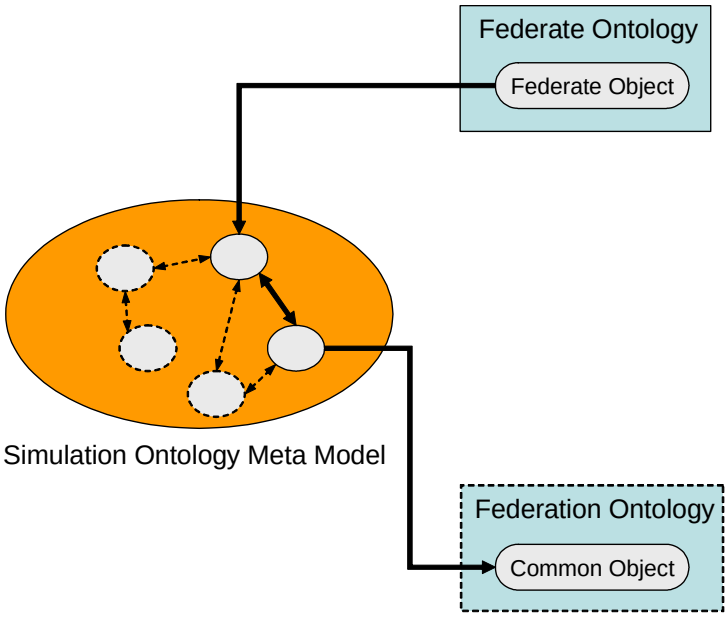


Figure 1.3: Composing a Chain of Relationships

The hypothesis proposed corresponding to research question 3 is:

Hypothesis 3: Relationships between federate simulation entities are captured in terms of a relationship with their common, federation-level representation. The relationships between concepts defined in the simulation ontology metamodel can be composed together to derive the federate-common entity relationships. An algorithm can be developed to generate a connected graph of existing relationships in the federation. Graph traversal algorithms can be developed to identify relationships between simulation entities a chain of these existing relationships.

It is important to note that not every relationship and associated transformation stub can be derived automatically. Only those relationships that can be expressed by applying existing semantics can be inferred. Even if all simulation entities are expressed in terms of the same baseline concepts, additional knowledge may be required to determine

relationships between two (or more) simulation entities. As discussed earlier, two coupled simulation entities may refer to different but related concepts (such as radius and diameter). While a relationship between their disparate representations (such as unit conversion) can be derived automatically, the inherent relationship between the two entities (`radius=diameter/2`) must be specified by a human. Moreover, the set of relationships defined in the ontology metamodel may not be comprehensive. Therefore, it is important to consider and provide for human interaction and knowledge input in the proposed framework.

Furthermore, it should be noted that in taking on the approach of composing existing transformations together, the automatically derived transformation stub may not always represent the *best* conversion between two simulation entities. The best transformation is the simplest composition of existing relationships that entails the least amount of information loss in converting a simulation entity from one representation to the other. The automated selection of transformations is based on a heuristic that may not always report the best possible composition of relationships. Furthermore, the limited set of semantics captured in this framework does not allow us to specify degrees of information loss in transformations. Therefore, we are not able to distinguish between transformations that involve information loss. Finally, a composition of existing transformations may not always be valid; knowledge to establish the validity of a composed transformation is not captured in this framework. For these reasons, it is apt to use the approach outlined above to arrive at 'suggested' transformations between simulation entities in an automated

fashion. These suggestions may be approved or revised manually, based on whether the transformation arrived at is satisfactory.

The research questions posed above are directed towards alleviating the cost of achieving interoperability between federate simulations, specifically focused upon the issue representational compatibility. Question 1 is a rather broad question in which automating of the process of achieving representational compatibility is pondered. In the associated hypothesis, the use of semantic technologies to capture and apply knowledge in support of such automation has been elaborated. The vision for a framework in which the semantics of simulation concepts are described in ontologies and used to automate the process of relating federate simulation entities is presented. This vision leads into research questions 2 and 3 in which the specifics of representing simulation concepts in ontologies and the subsequent automated generation of transformation procedures are contemplated. In the context of the vision presented, the approach of capturing semantics in discrete layers and composing together existing relationships is proposed. In automatically defining relationships between disparate representations of shared simulation concepts, this framework offers significant potential to assuage the task of achieving technical interoperability in a distributed simulation. Furthermore, the ontology-based approach supports reuse of existing federate simulation models in myriad distributed simulations. Once a formal description of the objects and events represented in a given simulation model are captured in an ontology, that knowledge can be applied every time the simulation model is part of a new federation. That is, a set of

transformation stubs to convert to and from any federation's common representation could be arrived at in an automated fashion.

That being said, it is important to note that universal interoperability is not, and should not be the goal of this framework. Universal interoperability is the ability of a simulation to interoperate with any federation, regardless of purpose or technical implementation. Different simulations are developed with different purposes in mind. In many simulation models, behaviors and laws are approximated. Approximations made in one model may not be valid in another. The set of concepts defined in one simulation may be more superficial than in another. The integration of such disparate simulations could render the results of the resultant distributed simulation invalid or untrustworthy. Moreover, there are other issues related to time-management and run-time information exchange that could impact interoperability. The focus of the framework proposed is not to determine simulation compatibility or guarantee interoperability, but to simplify and support the process of arriving at an interoperable set of simulations.

The research questions and hypotheses developed in this section form a scaffolding for the remainder of this thesis—the subsequent chapters are focused on further developing, verifying and validating the vision presented above. In the following section, the strategy employed to validate the hypotheses is elaborated, following which, the organization of the remainder of the thesis is presented in the context of answering the research questions identified. Before proceeding to these sections, the research questions and associated hypotheses are collectively reiterated below in Table 1.1.

Table 1.1: Research Questions and Hypotheses

No.	Research Questions and Hypotheses
Question 1	How and to what extent can the process of achieving representational compatibility between simulations in a federation be automated?
Hypothesis	Ontologies can be used to formally describe the semantics of concepts in a federate simulation model. These semantics can then be applied to generate a required common information model and associated transformation stubs in a partially automated fashion.
Question 2	How should simulation concepts be represented in an ontology to support achieving interoperability?
Hypothesis	A metamodel for specifying simulation ontologies can be developed. The set of concepts and relationships between them defined in this metamodel form a vocabulary for describing simulation ontologies. If all simulation concepts are modeled using the same vocabulary, the relationships between two coupled simulation concepts in a federation can be inferred in an automated fashion.
Question 3	How can the transformations between two federate simulation entities be derived in an automated fashion?
Hypothesis	Relationships between federate simulation entities are captured in terms of a relationship with their common, federation-level representation. The relationships between concepts defined in the simulation ontology metamodel can be composed together to derive the federate-common entity relationships. An algorithm can be developed to generate a connected graph of existing relationships in the federation. Graph traversal algorithms can be leveraged to identify relationships between simulation entities a chain of these existing relationships.

1.3 Validation Strategy

The strategy employed to validate the work presented in this thesis is derived from the validation square developed by Pedersen and coauthors (Pedersen, Emblemsvag, Bailey et al. 2000). The validation square, originally developed to support validation of design methods, is a contextual process of demonstrating the usefulness of a design method in serving some purpose. Within this model, 'usefulness' of a design method is associated with both its ability to provide design solutions correctly (structural validation) and its ability to provide sound, correct solutions (performance validation).

The validation square is really a composition of four distinct parts, as illustrated in Figure 1.4. The validation process begins in the upper left-portion and proceeds in a counter-clockwise direction. The first quadrant deals with Theoretical Structural Validation wherein the validity of the individual constructs of the design method is accepted. The second quadrant, labeled Empirical Structural Validation deals with accepting the validity of the example problem(s) used to demonstrate the purpose of the design method. Next, Empirical Performance Validation is conducted, wherein the usefulness of the design method in the context of the example problem is accepted. The final component of the validation square, Theoretical Performance Validity involves building confidence in the generality of the design method, and its usefulness beyond the example problem. This entails building up confidence in the method based on the acceptance of prior structural and performance validity, based on which 'a leap of faith' is taken as to the general validity of the work.

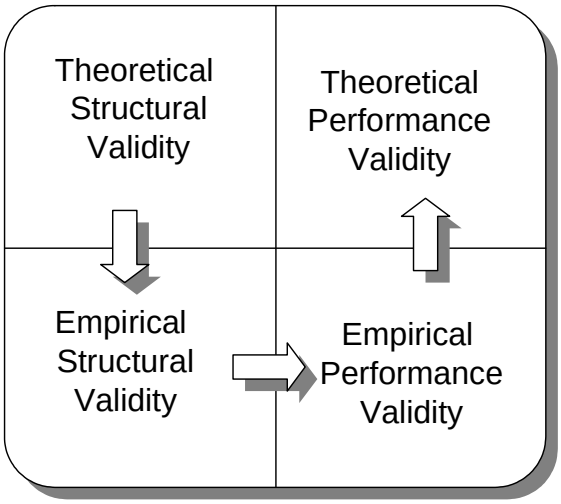


Figure 1.4: The Validation Square (Pedersen, Emblemsvag, et al., 2000)

The validation square is leveraged to validate the ontology-based framework for integrating simulations in a federation. Each quadrant of the validation square is addressed in this thesis, following the process indicated above. First off, the key aspect in accepting the Theoretical Structural Validity of the framework is to determine ascertain that it is based on a sound foundation. In Chapter 2, a survey of existing work that can be leveraged in the development of this framework is presented. This literature survey helps to determine the internal consistency of the framework and the individual methods and constructs it makes use of. In Chapter 3, a process model indicating how individual constructs in the framework come together to support the end purpose (achieving interoperability in an automated fashion) is used to determine the soundness of the framework as a whole. To address Empirical Structural Validity, the appropriateness of the example problem to demonstrate the intended use of the framework is discussed in Chapter 5. The ontology-based framework is applied to the development of an air traffic

federated simulation, which is representative of a variety of representational inconsistencies that this framework is geared to address. The usefulness of the framework in the context of its application to this example problem, i.e. Empirical Performance Validity is also ascertained in Chapter 5. Here, a detailed discussion as to if and how the framework is useful in achieving interoperability between the three simulations that comprise the air traffic federation is included. Finally, Theoretical Performance validity of the framework is addressed in Chapter 6, where a generalization of the frameworks usefulness beyond the example case is discussed. The strategy for validating this work in the context of the validation square is present below in tabular form (Table 1.2). This table provides the reader with a validation roadmap indicating where and how each quadrant of the validation square is addressed in this thesis.

Table 1.2: Strategy for Validating the Work Presented in This Thesis

Aspect of Overall Validity	Strategy Employed	Corresponding Chapter
Theoretical Structural Validity	Conduct literature review to determine the basis and soundness of framework constructs	Chapter 2
	Create overall framework process flow model to ascertain internal consistency of complete framework	Chapter 3: *Section 3.7*
Empirical Structural Validity	Discuss example problem background. Show that example problem is within the range of intended use of the framework.	Chapter 5: *Section 5.2*
Empirical Performance Validity	In the context of the example problem, determine if the framework does 'what it is supposed to do'. Discuss the merits of using the framework to mitigate the cost of achieving interoperability between constituent simulations of an air traffic federation	Chapter 5: *Section 5.6*
Theoretical Performance Validity	Make a leap of faith based on structural and performance validity accepted thus far.	Chapter 6

1.4 Organization of Thesis

Having presented the strategy to validate the usefulness of the ontology-based framework, this chapter is closed with a brief description of how the rest of this thesis is laid out. The organization of the thesis is illustrated in Figure 1.5, which is a modification

of the thesis roadmap developed in the work of Seepersad (2001). This figure indicates the development and testing of the hypotheses posed and is meant to guide the reader through the development and validation of the research work documented in this thesis.

In Chapter 1, the context and motivation for the research conducted has been elaborated. The overall research question and its two subordinate questions, and their respective hypotheses are posed, thus setting the foundation for the rest of the thesis. A survey of related work in the realm of federated simulation and information model management is presented in Chapter 2. In Chapter 3, the ontology-based framework for supporting automation in simulation integration is detailed. The development of the individual components of this framework is elaborated here. Following this, the process by which knowledge captured in ontologies is applied to automatically instantiate transformations between simulation entities is presented in Chapter 4.

The following chapters address the performance validity of framework outlined in Chapters 3 and 4. In Chapter 5, an example federation development problem associated with the design of a complex airport system is presented, and the use of the framework to integrate a set of federate simulations (namely the air traffic, ground traffic and ground services federates) is demonstrated. Finally, Chapter 6 concludes this thesis with a summary of work, a critical review and a set of recommendations for further investigation and extension of this research.

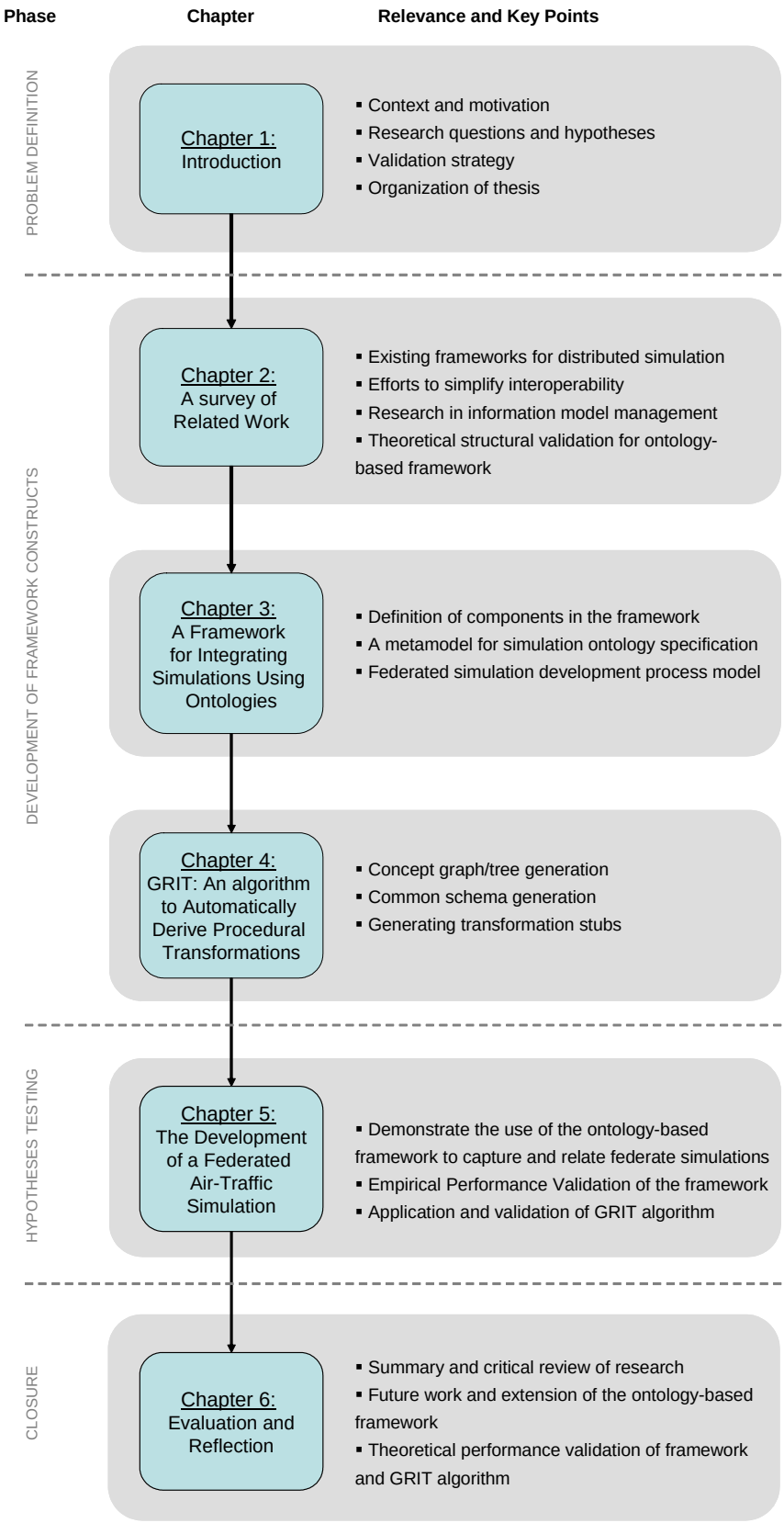


Figure 1.5: Thesis Roadmap (based on Seepersad, 2001)

CHAPTER 2

A SURVEY OF RELATED WORK

The purpose of this chapter is to review and assess existing research and development work that is pertinent to answering the research questions posed in Chapter 1. Given the overall goal of simplifying and supporting the simulation integration process, two key research areas have been surveyed. First, an existing framework to support federated simulation, the High Level Architecture (HLA), is studied with the emphasis on how information modeling has been used to support the task of developing simulation federations. Within the HLA framework, we explore existing systems to support automation and reuse in the federation development process, which provide insight into the development of the proposed ontology-based framework.

The second component of this literature survey is focused in the realm of information model and schema management. In Chapter 1, we proposed that ontologies can be used to capture simulation domains and relationships between them. Significant research has been conducted to address this very issue i.e. how to manage information stored using distributed and disparate models. Hence, in Section 2.3, general frameworks developed to address ontology and schema management are discussed. These frameworks provide a solid foundation for the development of an ontology-based framework where in simulation concepts can be captured and related across different domains.

2.1 Federated Simulation in the HLA

The High Level Architecture (HLA) is an architecture for federated simulation, which can be used by simulation developers and users to create simulation applications (Dahmann, Fujimoto and Weatherly 1997; Dahmann, Salisbury, Turrell et al. 1999; Defense Modeling and Simulation Office (DMSO) 1999; Kuhl, Dahmann and Weatherly 1999). HLA is an entire framework that is used to model federate simulations, group them together in federations and facilitate the execution of federated simulations wherein information can be exchanged between federates in real-time. Since the capture and use of simulation information models in HLA is similar to our approach to support simulation integration (as discussed in Chapter 1), it is important to understand how information models are used to help develop federated simulations in HLA. Much of this approach can be leveraged in implementing the proposed framework. The process of developing HLA federations, associated limitations and existing work undertaken to address those limitations are elaborated in this section.

"The High Level Architecture (HLA) is a general-purpose architecture for simulation reuse and interoperability. The HLA was developed under the leadership of the Defense Modeling and Simulation Office (DMSO) to support reuse and interoperability across the large numbers of different types of simulations developed and maintained by the Department of Defense" (Defense Modeling and Simulation Office (DMSO) 2004) . HLA emerged as a result of three Defense Advanced Research Projects Agency (DARPA) contracts in 1995, and since then it has grown to become the preferred

architecture for simulation interoperability within the department of defense, and is an open standard of the Institute of Electrical and Electronic Engineers (IEEE) (IEEE 2000).

Functionally, the HLA consists of two major components. The first of these is the set of federate simulations in a given HLA federation. HLA generalizes 'federates' to include manned simulators and human participants, as is often the case in many defense-related simulation scenarios. The second component in the HLA is the interface between federates and a Run Time Infrastructure (RTI). The RTI is an operating system for distributed simulation, which "provides facilities for allowing federates to interact with each other and is a means to control and manage the execution (of an HLA federation)" (Fujimoto 2000). The HLA interface specification defines a standardized way in which any federate interacts with the RTI. These functional components are illustrated in Figure 2.1.

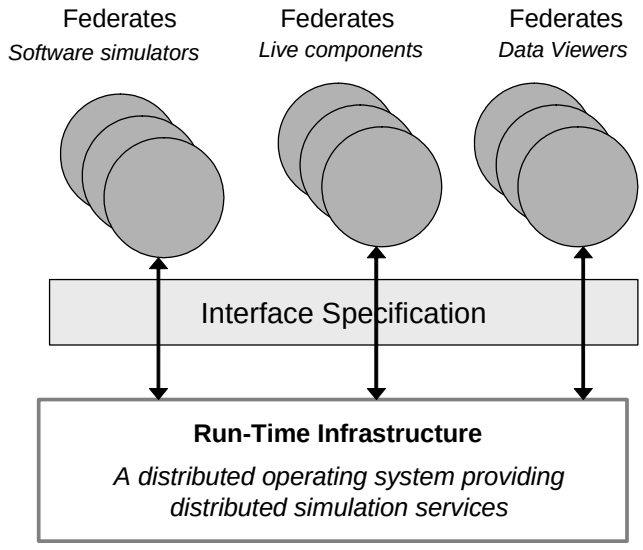


Figure 2.1: Functional Overview of the HLA (Fujimoto, 00)

The HLA is formally defined by three IEEE specifications—(i) HLA rules (IEEE 2000) (ii) the object model template (OMT) specification (IEEE 2000) and (iii) an interface specification (IEEE 2000). HLA rules describe the key set of principles upon which HLA is based. These rules define what the different HLA federation components are responsible for and how they interact at a high level of abstraction. The object modeling component specifies the information structure and representation of all shared information in a federate or federation. This includes the set of all concepts (objects), their attributes and their associations (interactions). Finally, the HLA interface specification is a description of the interface between federate simulations and a Run Time Infrastructure (RTI).

The HLA framework is geared to facilitate message-based interaction between federate simulations. Through the interface specification, a federate simulation may share information with its counterparts by employing the *publish* service provided by the RTI. The RTI then transfers the published message to other federates by means of a *subscribe* service. For a given published simulation entity, knowledge as to which federates are to be notified of that message (i.e. the subscribing federates) are documented in HLA object models, which are discussed in detail below.

2.1.1 HLA Object Models

HLA Object models are information models which represent the concepts in a simulation domain. These models are used to specify the representation of shared concepts in a federation, which is to be reflected in each underlying federate simulation. “The HLA is

directed towards interoperability; hence in the HLA, object models are intended to focus on descriptions of the critical aspects of simulations and federations which are shared across a federation" (Dahmann, Fujimoto and Weatherly 1997).

Object models are defined in table format, as specified in a meta-model, namely the OMT. The OMT defines tables for specifying simulation Objects (persistent concepts in a simulation) and Interactions (non-persistent, transient concepts). Object tables are used to specify a hierarchy of classes of objects in a federation and capture related meta-data about objects (such as name, purpose, version etc). Each object can have a set of attributes associated with it, defined in an attribute table. Similarly, Interaction tables define the HLA Objects involved in a given simulation occurrence (stating whether the objects are initiators or reactors in the interaction) and can have parameters associated with them that are defined in a parameter table. Attribute and Parameter tables define different characteristics of attributes and parameters, respectively, such as their data type, cardinality, resolution, units and update type (periodic or conditional).

The HLA OMT provides insight into one way in which concepts in a simulation domain can be represented in an information model. This template can be leveraged in the definition of a metamodel for capturing a simulation domain in an ontology. Although the table structure specified in the OMT differs from the frame-based representation employed in ontologies (Lassila and McGuinness 2001) (and their serializations (World Wide Web Consortium (W3C) 2004)), the tables can be used to define the concepts of

objects, interactions, attributes and parameters, their properties (table fields) and relationships.

There are two types of HLA Object Models defined using the OMT. These are Simulation Object Models (SOMs) and Federation Object Models (FOMs). A SOM, associated with a federate simulation, is meant to document the set of concepts as they are represented in that simulation. This documentation is used to gauge if the simulation is appropriate for participation in a given federation. A FOM specifies the set of shared information in the federation, clearly defining their representation in that federation. The FOM and SOM are instrumental in the realization of a functional federated simulation, elaborated in the FEDEP. The following section is focused on the FEDEP, indicating how FOMs and SOMs are used in this process.

2.1.2 Challenges in Federation Development

The HLA Federation Development Process (FEDEP) model is a systems engineering model for federation development (Defense Modeling and Simulation Office (DMSO) 1999). The FEDEP also serves as a common reference model for distributed teams developing federations, and provides a mechanism to order and share federation development experiences (Lutz 1999). The FEDEP employs a multi-step process model that guides federation developers through the development of a federated simulation from conception to testing and verification. It is important to study this federation development process as it elaborates (to a significant level of granularity) individual steps involved in

integrating simulations based on their information models. Using this model, the potential for automation at each step can be identified and addressed.

The FEDEP model, illustrated in Figure 2.2, is a six-step process for federation development. It is not so much an iterative process, but involves feedback and refinement of individual steps as the federation development proceeds. Much of the FEDEP focuses on the development of the FOM for a given federation and the selection of an appropriate set of federates. Steps 1 and 2 deal with the development of a conceptual federation model and the creation of a requirements list for the federation, known as the federation blueprint. The next two steps deal with the design and development of the conceptual model, where federate selection and FOM creation take place. Finally, steps 5 and 6 deal with the actual integration of federates and federation testing.

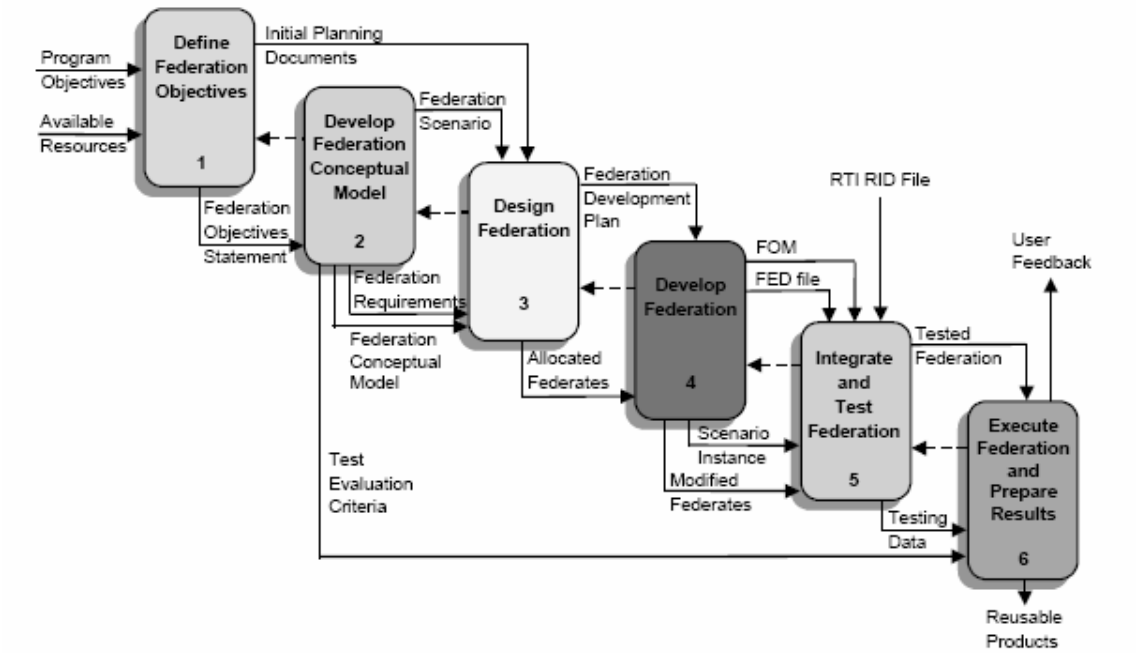


Figure 2.2: The Overall FEDEP Flow Model (DMSO, 99)

HLA's solution to dealing with multiple representations of shared concepts is to impose one global representation, which is defined in the FOM. This is evident in steps 3 and 4 of the FEDEP. Federates are selected based on their ability to meet the requirements defined in the federation blueprint. The FEDEP model then suggests several approaches to developing a FOM including a 'bottom up' design from scratch, a 'merging' of participating SOMs and modifying an existing reference FOM. The latter methods are efficient in that they leverage existing object models. Once the FOM has been defined, the issue of representational consistency is tackled. As illustrated in Figure 2.3, each federate simulation's code is to be modified so that the resulting SOMs and target FOM are consistent. In other words, all federates have to conform to the 'common' representation for the full set of exchangeable data that is defined in a FOM. Therefore, in a given HLA federation information exchanged during run-time cannot have disparate federate representations.

The efficacy of implementing reuse in HLA is marred by the fact that cost and time required to achieve reuse are strongly affected by the uniformity of the federate representations. Reuse is an important directive of the FEDEP—the process of developing a federation from scratch is complex and resource intensive; hence it becomes important to leverage existing work where possible. To this extent, HLA relies on the use of 'reference' FOMs as a starting point for developing new federations. Still, the fact that a simulation model has to be modified every time is participates in a new federation indicates that this approach does not currently support a great extent of reuse. The importance of reuse in the FEDEP has been acknowledged in the community and efforts

have been made to ameliorate reusability in HLA (Scrubber, Lutz and Dahmann 1998; Turrell, Bouwens and McCormack 1999). It is important to review this work, as it can be leveraged in answering the research questions posed in Chapter 1.

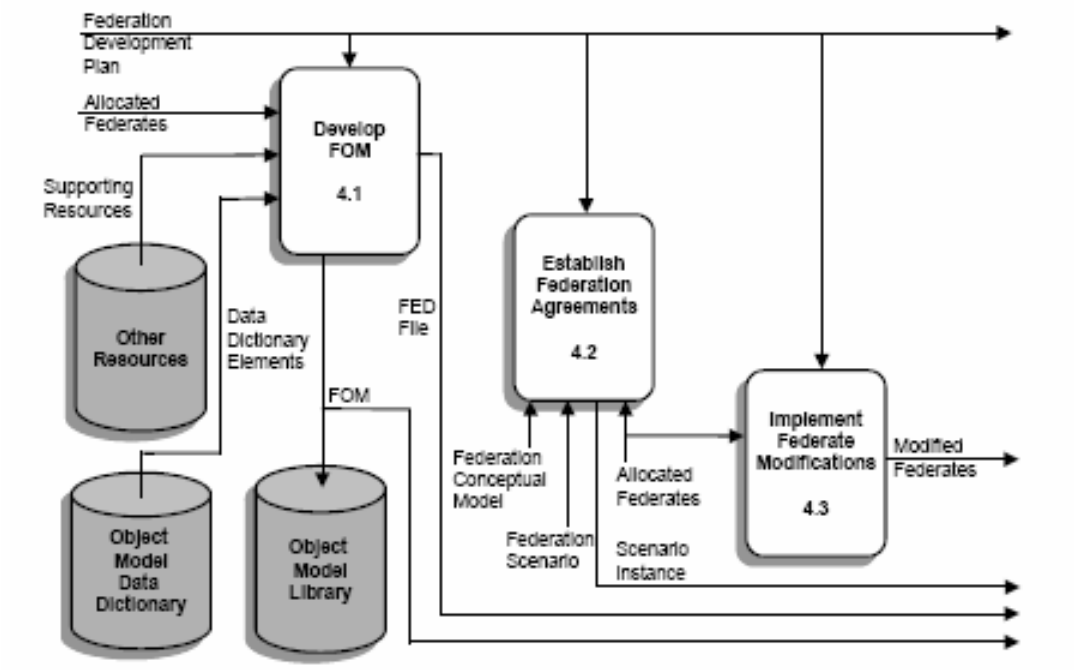


Figure 2.3: Detailed View of Step 4 in the FEDEP (DMSO, 99)

The issue of reuse in HLA corresponds directly to the motivation for the research presented in this thesis. The task of instantiating relationships between disparate representations of shared concepts in a federated simulation requires significant time and effort. Performing this task in an automated fashion, reusing existing work where possible, would have a sizeable pay-off. A system to support the automation of such tasks must reuse existing (formalized) knowledge. Since reuse is key to automation, the work

done to address the reuse in HLA federation development has a strong relationship to the research goals of this thesis. Hence, a detailed discussion of existing frameworks developed to facilitate reusability in HLA follows in the sub-sections below.

2.2 Current Solutions to Support Reuse in Federation Development

In this section, existing work to improve the efficiency of federate reuse in HLA federations is detailed. Specifically, the Base Object Model framework (Gustavson, Hancock and McAuliffe 1998) and the Agile FOM Framework (Macannuco, Dufault and Ingraham 1998) are discussed. The goal of this discussion is to identify the salient points of these frameworks and how they relate to a framework to support federate simulation integration. The ideas developed to support HLA federate reuse can be generalized and leveraged to address the issue of simplifying the process of establishing relationships between disparate representations of shared concepts in a federated simulation.

2.2.1 Base Object Models

The goal of the BOM framework and the ontology-based framework we have proposed in Chapter 1 are similar; however, the approach taken is distinct. The Base Object Model (BOM) framework has been developed as a means of simplifying the federation development process and supporting the reuse of existing object models in HLA federations. BOMs are reusable building-blocks to construct federate and federation information models.

The HLA FEDEP states that FOM reuse and integration through piece-parts is the most desirable method for federation construction (Defense Modeling and Simulation Office (DMSO) 1999). This piece-part approach to FOM development should involve exploring the reuse of existing SOM and FOM piece parts in federation development. This idea has been leveraged in the development of BOMs, with the goal to improve reusability and enable rapid federation development. At the outset, it is evident that the goal of the BOM concept and the framework being presented in this thesis are geared towards achieving the same goal, namely to simplify the process of achieving interoperability between federated simulations. In this research, the approach is to employ knowledge reuse to support the federation development process. Similarly, the BOM concept has been developed to capture and use meta-data to simplify the FOM development process, as explained below.

A BOM is defined as a simulation component that represents a single aspect of simulation interplay in a FOM (or SOM) that is used as a building block for FOM and SOM specification (Gustavson, Hancock and McAuliffe 1998). In other words, a BOM is subset of a FOM (often referred to as a mini-FOM) in which a portion of the overall interaction between federates is captured. This concept can be viewed to be analogous to a LEGO block (the BOM), several of which together can be used to form a number of different structures (the federations) (Base Object Model Study Group 2001). Based on concepts defined in the HLA OMT, a BOM consist of several Objects, Interactions and associated attributes and parameters, respectively. In addition, a BOM includes meta-data

describing the simulation interplay aspect it models. The general BOM structure is illustrated in Figure 2.4.

A BOM represents simulation interplay by an interaction class and the set of objects involved in that interaction. In the case that one set of involved objects responds to a stimulus arising from another participant object (the interaction is not reciprocal), a 'trigger' BOM is employed. For bi-directional interaction, another class of BOMs is defined, namely the 'interaction' BOM.

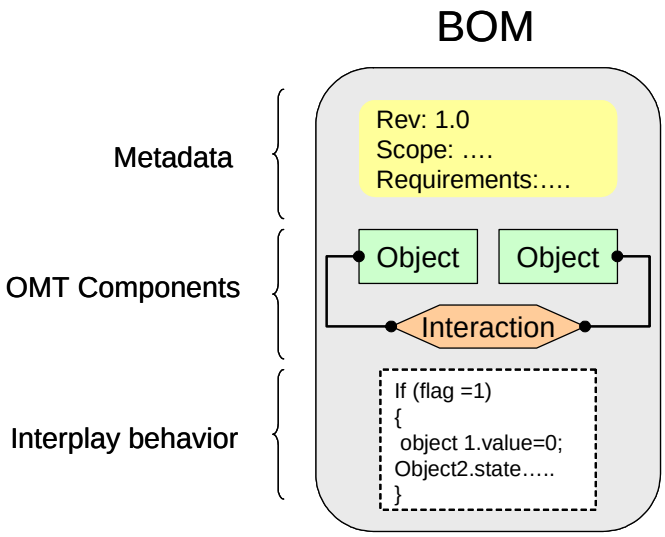


Figure 2.4: Components of a BOM

The key component in a BOM that enables its reuse (and possible automation) in FOM development is the meta-data it captures about a given interplay aspect it models. This meta-data (listed in Table 2.1) includes (but is not limited to) information about the

requirements, intended domain and scope of a given BOM, its conceptual model, a set of application scenarios and best practices on integrating the BOM in various FOMs. This information is useful in applying an existing BOM to new federated simulation scenarios, thus facilitating reuse. The requirements listed in the federation blueprint are used as parameters in a meta-data search through a repository of existing BOMs. If a match is found based on the information contained within a BOM's meta-data, that BOM can be integrated into the FOM being developed.

Table 2.1: BOM Meta Data Fields

Metadata Element	Sub-Elements
Requirements	
Conceptual Model	
Accreditation Information	
Intended Domain and Scope	
Integration Experience	Process
	Products
	Lessons Learned
	Use History
Revision history	
Graphics	2D/3D Models
	Textures
	Key Frames
Other	Sequence Diagrams
	Scenario Application

While it is not clear whether BOM meta-data is captured in a formal, machine-processable format (Base Object Model Study Group 2001) suggests that an XML

Schema should be developed corresponding to the HLA OMT, which could then be extended to capture information about meta-data (Miller and Filipelli 1999). Based on this, meta-data matching could be performed using the XML-Query Language (XQuery) developed by the W3C (World Wide Web Consortium (W3C) 2003). Potentially, the later steps of FOM development process could be automated using meta-data to (i) identify appropriate BOM's to meet requirements and (ii) integrate the BOMs to realize a FOM.

The BOM reuse methodology relates to the hypotheses posed in Chapter 1 in that it taken on the approach of meta-data capture to support reuse in the development of simulation federations. However, the type of meta-data and the subsequent use of that meta-data is distinct in the two approaches. BOMs use meta-data to document the intended use of a complete interplay component and thereby identify reusable components for the development of different FOMs. That metadata is then used to simplify the development of a global, federation-wide information model. While the BOM methodology alleviates the difficulties in arriving at a FOM, the task of modifying individual simulations such that they are consistent with the FOM is not addressed. In contrast, we propose to capture meta-data about individual concepts in federate simulations (rather than an entire simulation interplay scenario). Based on the semantics (meaning) of each concept, the relationship between two concepts could potentially derived at automatically. This not only enables automated FOM generation, it facilitates as-is federate reuse. The research presented in this thesis takes on this approach, thereby going beyond the functionality that BOMs offer.

According to the BOM study group, in the future, the BOM framework could be extended to facilitate the rapid integration of existing SOMs in disparate FOMs (Base Object Model Study Group 2001). If SOMs and FOMs were built using BOMs, a BOM level mapping between the two could be specified by identifying similar "patterns" in the structure of the SOM and FOM. To achieve this mapping functionality in an automated fashion, a richer set of BOM meta-data (semantics) would be required. A final limitation of the BOM framework is that creating and managing BOMs is a task that could potentially entail a large cost and time overhead. If, during FOM development, a BOM to match a set of blueprint requirements is not found, a new BOM needs to be created, which is not a trivial task. It involves modeling individual participants in the interaction, as well as the implementation of that interaction behavior (in a given programming language). Finally, all meta-data as to the scope of the newly created BOM etc. has to be generated. Managing a repository of BOMs in itself calls for a significant effort in the way of sorting, arranging and standardizing the contribution of new BOMs. Finally, it may not be plausible to maintain a central repository of all BOMs if their number and use were to multiply at a large rate. For these reasons, it is not feasible for supporting simplified integration in distributed simulation, in general.

2.2.2 The Agile FOM Framework

The Agile FOM Framework has been developed at Lockheed Martin Information Systems (Macannuco, Dufault and Ingraham 1998) to facilitate the integration of HLA federates wherein SOM and FOM representations do not have to be consistent.

As mentioned earlier, reusing a federate simulation in different HLA federations often entails modifying the underlying simulation model to be consistent with the representation defined in the FOM. The cost of performing this software engineering over and over again can be significant. With this in mind, the AFF has been developed to facilitate the as-is reuse of federate simulations in multiple federations. Rather than take on the approach of promoting reuse through standardization, the AFF aims at allowing simulations the freedom to maintain their own information representations. The foundational plot of the AFF is to map federate objects and interactions (and their attributes or parameters) to related entities in the FOM. These mappings establish relationships between different representations of shared concepts and are used to perform the conversions across disparate representations, to ensure consistent information transfer by the HLA RTI during execution. A conceptual view of the AFF is illustrated in Figure 2.5.

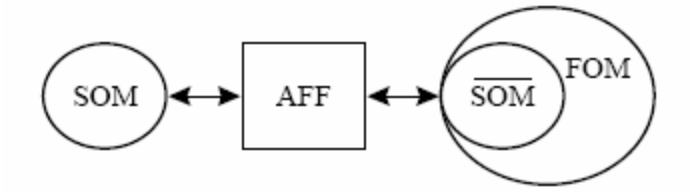


Figure 2.5: Conceptual Depiction of the AFF (Macannuco et al., 98)

The capabilities of the AFF were determined based on a study of a wide variety of SOMs, their potential use in various FOMs and the types of mappings that would have to be instantiated between the two to facilitate consistent information transfer (Macannuco,

Dufault and Ingraham 1998). One important feature of the AFF is its ability to deal with attribute atomic-ness. That is, the information stored in one attribute in a SOM may be represented with several attributes in a target FOM. The AFF's conversions provide functionality to split complex attributes apart or merge constituent attributes as dictated by a mapping. Further, conversions between representations employing different units of measurement, coordinate systems and byte arrangement systems ("little endian" versus "big endian") are supported by the AFF. Finally, the AFF has the capability to handle enumeration mappings, i.e., mappings between enumerated types where the enumerals (and number of enumerals) are not the same. It is important to make note of these capabilities as they indicate the types of relationships (and associated transformations) that exist between disparate representations of concepts in federated simulations. In the development of the framework proposed in Chapter 1, we consider how the types of relationships identified by the AFF developers can be captured in ontologies.

The key component for establishing mappings in the AFF is a converter. A converter is basically application-level code (procedures) that transform information from internal (SOM) to external (FOM) representations, and vice-versa. Converters not only capture the transformation from one representation to the other, but they interface with the HLA RTI directly to perform appropriate conversions when attributes and parameters are published or subscribed at run-time in a federation. The AFF identifies a set of basic properties that converters must satisfy in order for them to support real-time mappings. Two properties are of special interest—(i) Converters must be chainable and (ii) Converters must be bi-directional. The former indicates that relationships should be

reusable so as to apply them to develop new relationships. This property is fundamental in the knowledge reuse paradigm; to automate the instantiation of relationships between distributed simulation concepts, the use of existing relationships to infer new relationships is key. The latter indicates that a relationship between two representations of a shared simulation concept must encapsulate transformations going both ways i.e. from representation 1 to representation 2 and vice-versa. Both these properties have been incorporated into the ontology based framework presented in Chapters 3 and 4. An example AFF converter arrangement illustrating chaining and bi-directionality is depicted in Figure 2.6.

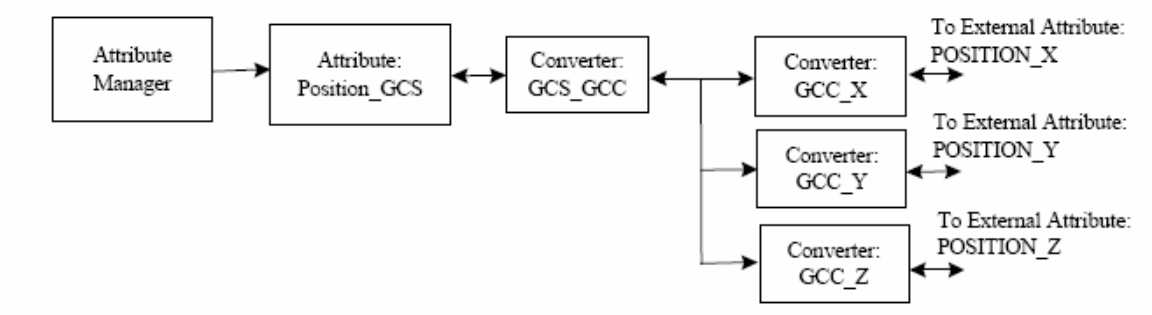


Figure 2.6: Example Converter Arrangement in the AFF (Macannuco et al. , 99)

The AFF is conceptually similar to the framework presented in this thesis. Both frameworks have the goal to simplify the process of integrating (reusing) federate simulations in multiple federations. Even though the scope of the AFF is limited to HLA federations, this work helps to identify some of the key issues that need to be addressed in implementing an ontology-based framework for relating federate simulation concepts in a federation. The types of possible relationships and the properties of the associated

conversions identified in the AFF are of specific importance. However this research goes beyond the AFF in that the focus is not only to enable disparate concept representations in distributed simulations, but to automate the process of relating these disparate concept representations. Aside from pre-defined data driven conversions such as unit transformations, all AFF converter procedures are specified by a human agent. Macannuco notes that in order to instantiate converters, knowledge of the SOM and a clear understanding of the representational differences between SOM and FOM is requisite. An ontology can capture this knowledge in a formal, machine-processable fashion, which could then be applied to automate the conversion generation process.

Having reviewed two frameworks that address reuse and automation in HLA federation development, the key points taken away from studying the HLA framework are summarized below:

- The HLA OMT provides insight into the development of a metamodel for capturing simulation concepts in an ontology.

- The HLA FEDEP prescribes the modification of federate simulation code to be consistent with the FOM. To facilitate a greater degree of automation, the ontology-based framework for integrating simulations in a federate should be able to integrate federate simulations ‘as-is’. One approach to do so is outlined in the AFF. Although the AFF does not automatically generate converters it may be possible to do so, as is explained in Section 2.2.

- The BOM framework uses meta-data to reuse piece parts in SOM and FOM development. The use of meta-data to support reuse and automation should be exploited in the framework proposed in Chapter 1. The piece-part approach to federation development itself calls for more overhead and is not viable for supporting automation in federated simulation outside of HLA.

The next section of this survey is focused on studying the highlights and limitations of commercial simulation-based design and analysis tools to support distributed simulation.

2.3 Simulation-Based Design and Analysis Tools

Having explored the HLA as a framework for federated simulation, we explore simulation-based design and analysis tools in this section. These tools do not support federated simulation; they are meant to be decision-support tools that are capable of performing system level simulation, analysis and optimization. System designers use these tools to connect models associated with different aspects (and sub-systems) of a system being designed in a serial fashion. This enables the designer to sequentially execute sub-system level simulations and analyze the behavior of the entire system. Essentially, these tools do not support any run-time interaction between simulations. However, relationships between parameters of different simulation models are specified using these tools. To that extent, it is important to study how these tools go about relating coupled simulation entities and to what extent this process is automated. Also, it may be possible to extend the applicability of our proposed framework to the simulation-based

design tool domain, which is another reason why we conduct this study. In this section, we focus on one simulation-based design environment, named ModelCenter.

ModelCenter, developed by Phoenix Integration (www.phoenix-int.com) is a software that has been developed to support model development and integration for engineering and simulation. “It illustrates the use of an integration architecture to meet the interoperability challenges faced by designers and analysts who are faced with the need to support acquisition decisions by using a distributed set of existing models” (Malone and Papay 1999). The fundamental component in this integration architecture is the Analysis Server. An analysis server is an encapsulation of a given simulation code (on a remote machine) such that it becomes a reusable module that can participate in any distributed simulation that is set up using ModelCenter. This wrapper includes (i) an information model that describes the various shared parameters and variables of a given simulation and (ii) an executable that serves the purpose of obtaining parameter values (and initial variable values) from specified files and running the simulation. Once wrapped, component simulation models can be connected to develop a federation in an intuitive fashion using ModelCenter’s GUI, as illustrated in Figure 2.7.

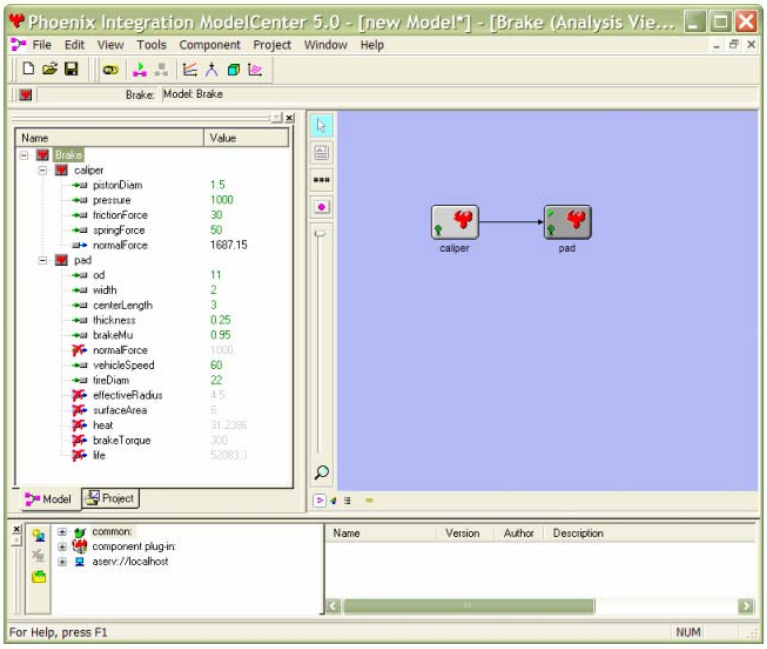


Figure 2.7: ModelCenter User Interface for Integrating Analysis Servers

A template for the analysis server information model is depicted in Figure 2.8. Information such as variable locations in the output/input files, their units (if applicable) and constraints on their values can also be specified. Variable hierarchies are not captured in this model, but can be defined in ModelCenter's GUI using script component objects. Using the metadata captured in these information models, ModelCenter has limited capability to map the different variables across disparate simulation components automatically. The 'Auto Link' feature is limited to detecting matches across linked simulation based on variable names or their positions in a variable hierarchy. Users may also instantiate links manually in a drag-and-drop fashion.

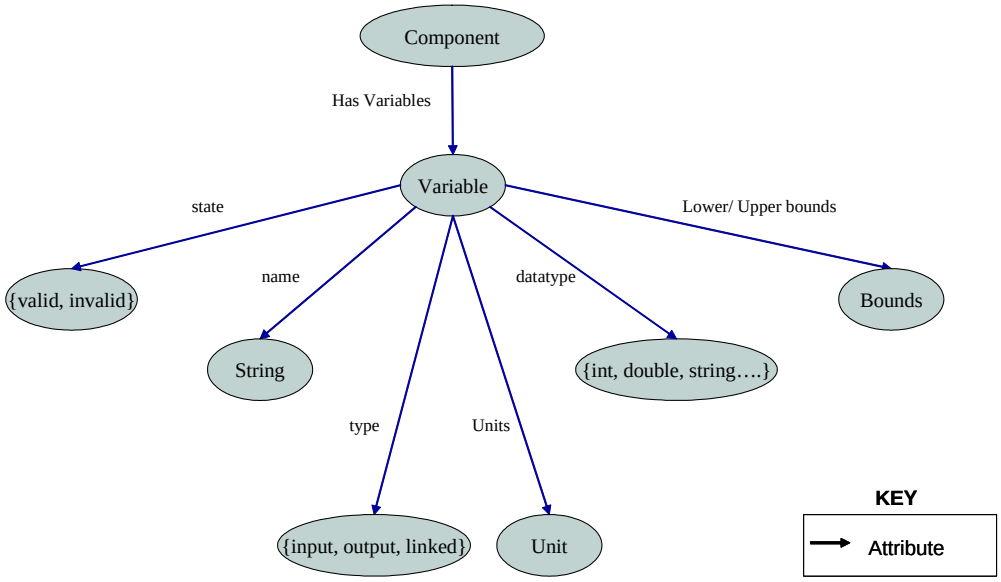


Figure 2.8: Analysis Server Information Meta Model

ModelCenter does not have the capability to identify representational inconsistencies between linked entities. All transformations between linked variables, such as unit conversions, must be specified manually by the distributed simulation developer. Ultimately, the use of a simplistic information model limits ModelCenter's ability to automatically identify and instantiate inter-domain mappings in a distributed simulation, outside of unit transformations. Based on the premise that mappings can be automated if the semantics of an entity are captured in a formal manner, a more expressive information model is required to define shared variables wrapped in an Analysis Server.

Furthermore, ModelCenter's functionality for running distributed simulation is limited to running participating simulations discretely. That is, there is no exchange of data between different analysis servers during the execution of a component simulation model.

Interoperability is limited to each server executing its code given a set of inputs (possibly from another server) and then providing its output as input to another server. Two servers can execute their code in parallel if they do not require input from each other (or the input of one is not determined by the output of the other, directly or indirectly).

The ModelCenter software tool is representative of most software environments developed to support simulation-based design and analysis. With regards to (the automation of) model integration, ModelCenter's highlights and limitations are indicative of most others. The research conducted in this thesis can be applied to alleviate some of these limitations. An automated approach to connecting component models to perform system-level simulation can significantly reduce the time and effort designers invest in such activities. In the next section, existing work related to the automation of the concept matching and mapping is visited.

2.4 Models and Algorithms to Manage Disparate Information Models

In the hypotheses posed in Chapter 1, ontologies have been identified as an avenue for capturing simulation domains and supporting the automation of relationship definition between shared concepts in a federated simulation. This is really a specialization of a general problem in database schema and ontology management, namely: relating distributed databases and ontologies with overlapping domains (Berners-Lee, Hendler and Lassila 2001). The use of databases and other computer-based repositories are ubiquitous in academia and industry. For example, in the realization of engineered products, databases are used to store product related data ranging from concepts,

geometry to manufacturing and disassembly. It is likely that this data, stored in disparate databases, each of which employs its own schema, will have to be integrated to study an existing product's realization or to perform product modifications and adaptations. To share such information across different sources, it is essential to address the issue of mapping between different information representations. This issue is pertinent in the domain of semantic technologies as well. Ontologies play a prominent role in publishing data on the semantic web. Given the distributed nature of the World Wide Web, it is likely that information on it will be captured in myriad ontologies. Therefore the task of establishing semantic mappings between ontologies is an important one.

This issue of relating disparate information models is being researched by several groups, both in the domain of ontologies and databases. Several researchers have developed fundamental theory and models for establishing and representing relationships between different information schemas (Alagic and Bernstein 2001; Madhavan, Bernstein, Domingos et al. 2002; Maedche, Motik and Stojanovic 2003). Others have used this fundamental basis to develop algorithms and frameworks that support automating the information model relationship process (Noy and Musen 2000; Doan, Domingos and Halvey 2001; Madhavan, Bernstein and Rahm 2001; Peak 2003). However, most existing frameworks are focused on the task of automating the task of schema/ontology *matching* (Milo and Zohar 1998; Madhavan, Bernstein and Rahm 2001; Rahm and Bernstein 2001)—the task of determining which elements of two or more schemas are equivalent. While the matching task is an important one in integrating federates in a federation, there is still the issue of *mapping* i.e. determining the representational transformations between

two matched entities. The research contribution in this thesis addresses the automation of this process.

Nonetheless, a lot of this existing work can be leveraged in the development of a framework to support automated mapping between federate concepts in a distributed simulation. Specifically, existing models for representing mappings can be applied to determine an underlying meta-model for adequately representing relationships between federate representations of shared simulation concepts in an ontology. The algorithms and frameworks that address automating the matching process can be leveraged in the development of an algorithm to support the process of instantiating mappings between federate ontology entities. In the following sections, the highlights and shortfalls of the above-mentioned existing contributions are detailed.

2.4.1 Models for Schema and Ontology Management

In this section, related work in the area of schema management is discussed. Several researchers have been addressing the problem of managing and integrating information distributed in databases and repositories employing different schemas. To solve this problem, several formal, generic models and frameworks for maintaining correspondences across different schemas have been proposed. Two such models, representing the majority of work in this area, are discussed below. These models provide insight as to what key features must be embodied in a framework to support the integration of participants in a federated simulation.

A theoretical framework for managing model meta-data has been developed by researchers at Microsoft (Bernstein 2003). This framework specifies a generic model for managing meta-data, applicable to many different modeling environments such as UML (Naiburg and Maksimchuk 2001), XML schema (Walmsley 2001) and Enhanced Entity Relationship Diagrams (EER) (Chen 1976). A general baseline model for representing meta-data is prescribed, so as to be at least as expressive as EER models. The model defines the existence of objects, properties and relationships (aggregation, generalization and asscociations). Based on this model, as set of model-management operators have been defined, namely Match, Compose, Dif and Merge.

Two operators of special interest are Match and Compose. A Match signifies the existence of a mapping between two entities. A mapping is defined as a set of objects that relate two matched objects. The actual relationship between the schema objects and the associated mapping is captured in a schema morphism (Alagic and Bernstein 2001). In other words, a mapping between two schema objects (as defined in this framework) consists of an intermediate representation of those objects and a set of data translations between the source/target and intermediate representations. An example of such a mapping, between two schema objects that capture information about employees is illustrated in Figure 2.9 (Bernstein 2003). This approach to relating concepts across different information models has two basic advantages. First, the use of an object structure in the mapping is more expressive than the definition of relationship pairs (such as <*Name*, *FirstName*> and <*Name*, *LastName*> in the example depicted in Figure 2.9. If such relationships are defined directly, the structure of the relationship, embodied in

Map_{ee}, is lost.). Furthermore, the use of an intermediate representation means that new mappings can be instantiated to and from matched objects by defining a morphism to the existing intermediate object structure (A new object can be mapped to *Emp* or *Employee* by defining a morphism to Map_{ee}). The compose operation creates new object mappings by combining two existing mappings. The idea is to use knowledge of existing schema relationship to identify new ones. A generic schema for composing mappings has been developed.

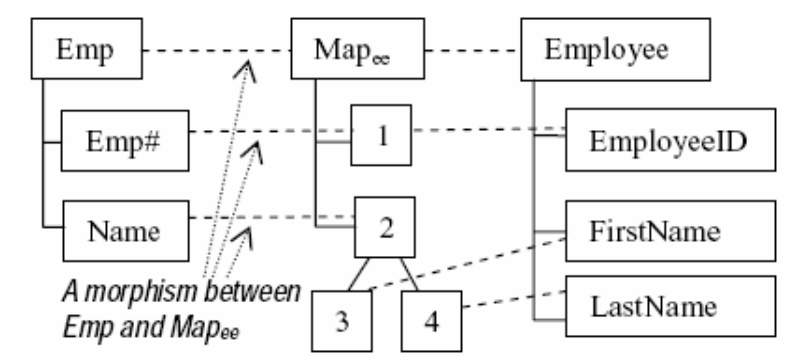


Figure 2.9: An Example Mapping Structure and Morphism (Bernstein, 03)

This framework for model management can be used as a foundation for defining relationships between federate simulation concepts in a federation. The representation of mappings via an intermediate set of objects and a set of morphisms can be leveraged to capture relationships and transformations between disparate representations of simulation concepts in an ontology. Based on the advantages that this approach offers, a more efficient method for relating simulation concepts can be realized. Furthermore, the composition operation can also be implemented in a system to support the generation of

transformations between federate concept representations. The definition of new mappings through composition results in a greater extent of automation; hence a similar approach has been embodied in the algorithm to generate representational transformations, which is detailed in Chapter 4.

Maedche and co-authors (2003) have developed a framework for managing multiple distributed ontologies wherein an ontology representation model and reuse system is defined. The object-instance (OI) conceptual model for defining, reusing and evolving ontologies defines the set of entities in an ontology and rules and relationships between them. These entities include concepts, properties, instances and structural relationships including cardinality, subsumption, property domain and range. Based on this metamodel for ontology specification, the issues of including distributed ontologies and evolving them are addressed. The approach to establishing connections between distributed ontologies covering overlapping domains is to include copies of an ontology in other related ontologies. Relationships can be instantiated (from the set defined in the OI model) to connect included and including ontology entities. Finally, a method to propagate changes in one ontology to all dependent ontologies (those that include a copy of the source ontology) has been developed.

The OI model defined in this framework can be leveraged in the development of a metamodel for simulation ontology specification. Equally important is the idea of including distributed ontologies to form an information model that spans a larger domain of discourse. In order to capture mappings between federate simulation ontologies, a

similar approach could be taken where all federate ontologies are included in a larger (federation) ontology. Having done so, relationships could then be specified between shared federate concepts.

However, the limited set of relationships defined in the OI model (domain/range and subsumption) is a significant restriction of this model. The framework is focused on *aligning* an included ontology to fit within the structure of an existing ontology. Ontology alignment refers to the task in which additional ontologies are 'fit' within the structure of an existing ontology. For example an ontology about passenger cars can be aligned to fit within a more general ontology about vehicles. Alignment does not address the issue of mapping between disparate representations of concepts. Consider the example of relating the concept of *address* having data type string to the concept *zipcode* of type integer in an included ontology. While it is intuitive that a *zipcode* is a subset of an *address*, the two concepts seemingly employ incompatible representations and their relationship cannot be defined based on domain, range or sumbsumption. Such scenarios are likely to occur in a federated simulation, and call for a more expressive model for relating ontology entities. Such a model is presented in the framework presented in Chapter 3.

2.4.2 Schema and Ontology Matching Algorithms

Having discussed existing models for representing relationships across disparate domains, the remainder of this section is focused on presenting existing algorithms and tools that support the automation of the schema matching process. Several algorithms have been developed, based on different approaches, to address the issue of identifying

matches. Algorithms like SEMINT (Li and Clifton 2000) perform matches using the instance pool associated with a schema, while others like SKAT (Mitra, Wiederhold and Jannink 1999) perform rule-based matching based on schema level information. Some systems such as LSD (Doan, Domingos and Halvey 2001) perform only 1:1 matches while other are able to handle *n:1* (and *1:n*) schema matches. Finally there are those algorithms that employ a single matching criterion as opposed to hybrid matchers that use multiple matching criteria. In this section, three such algorithms are discussed: PROMPT, an algorithm for ontology merging and alignment, GLUE an ontology matching system and CUPID, a generic schema matching algorithm. Note that automated matching only addresses a subset of the federate simulation integration problem. While the automated generation of data translations between shared concepts is not covered in these algorithms, much of this work is very relevant to the research conducted in this thesis. In the sections below, the salient features of the above-mentioned algorithms and their applicability to this research are highlighted.

Researchers at Stanford Medical Informatics have developed PROMPT: an algorithm and tool to support the automation of ontology alignment and merging (Noy, Fergerson and Musen 2000). This algorithm is based on the frame-based representation paradigm (Minsky 1975), very similar to the OI model presented above. A human-in-the-loop approach has been taken in the development and implementation of PROMPT. Developers acknowledge that it is not possible to completely automate the process of relating ontologies. Instead, the human is prompted to provide knowledge when required,

while the algorithm automates underlying tasks. The PROMPT algorithm flow is illustrated in Figure 2.10.

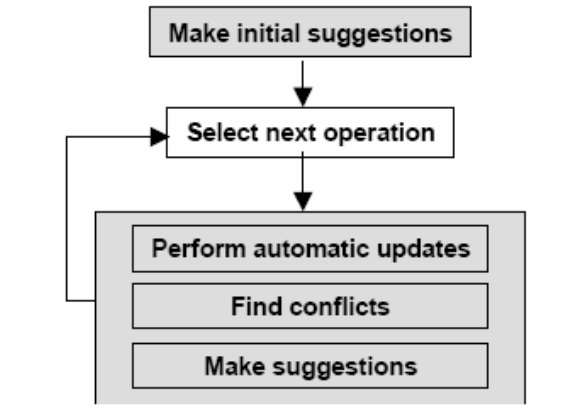


Figure 2.10: High-Level Illustration of the PROMPT Algorithm (Noy, 00)

To support automated ontology relation, PROMPT identifies concept matches across specified ontologies and reports them as suggested matches to a user. The matching criterion varies based on whether the intended task is to align two ontologies or merge them (amalgamate two sets of concepts into a single hybrid set). As the user approves suggestions, appropriate new entities are created in a target ontology and a new set of suggestions is reported. This is where the algorithm takes advantage of semantics to support additional automation. Based on the existing set of approved operations, and the set of concepts related to the matched entities, new suggestions are inferred. For example, if two concepts match, it is likely that their super classes will also match. Such automated inferencing means that matches are not identified solely based on linguistics. This means that users do not have to specify additional relationships explicitly, but instead just

approve them as they are reported. This method significantly reduces the time, effort and error-making associated with ontology alignment and merging (Noy and Musen 1999). Finally, PROMPT also uses semantics to determine conflicts (inconsistencies in the knowledgebase, such as the data type of a merged property) and suggest operations to solve them.

PROMPT does not address the issue of mapping disparate representations of related entities. As stated above, PROMPT is developed to address merging and alignment. Still, the tool represents one way in which automation can be supported in defining relationships across domain models. The same approach of suggesting mappings between related simulation concepts (how disparate representations translate) for users to approve or reject could be employed to support federate simulation integration.

Another tool that performs automated ontology matching, called GLUE, has been developed by researchers at the University of Washington. GLUE is a software system that employs machine learning techniques to match concepts across ontologies using multiple measures of similarity. This is in contrast with most other schema/ ontology matching systems that employ a single measure of similarity. Similarity measures are clearly defined so that there is an unambiguous understanding as to what is meant by a 'match' between two ontologies. These similarity measures (exact, most-specific-parent and most-general-child) are defined in terms of a joint probability distribution. GLUE uses a populated knowledge-base to determine a match between two given concepts. That is, from the pool of instances of concepts A and B, the set of instances $A \cap B$ is identified.

More accurately, the probability P (A, B) is calculated and used as the basic measure to determine whether A is similar to B, for a given definition of similarity. In order to determine the required probability, a multi-strategy learning system is employed. A content learner exploits frequencies of words in the textual content of instances A, B to determine P (A, B). A second learner, called the name learner calculates the required probability based on instance names. The predictions of individual learners are combined using a meta-learner to determine the final probability.

A major limitation of GLUE is that it makes heavy use of instances to determine matches. While this might make sense for any populated knowledgebase, it is not so in the case simulation information models. In a simulation information model, there are no instances—instances are created when the simulation is executed. An information model for a simulation usually captures concepts, their properties and relationships. Therefore the instance-based probability approach to determining entity matches cannot be employed to support simulation integration.

A more generic schema matching algorithm, called CUPID has been developed with the idea of combining the matching strategies employed by several existing schema matching systems (Madhavan, Bernstein and Rahm 2001). CUPID discovers schema matches based on names, data types, constraints and schema structure. Like GLUE, this algorithm employs a similarity coefficient (between 0 and 1) as the measure of the degree of similarity between two schema entities. However, similarity measures are not based on the instance pool. Instead, a multiple phase match strategy is used to calculate similarity

coefficients. The first phase calculates matches based on linguistics of schema entities and even employs a thesaurus to identify synonyms and short-forms. The second phase is a structural match, wherein the context of a schema entity and its vicinity to other matched entities are taken into consideration. This is similar to the method in which the PROMPT algorithm identifies new operations based on existing approved ones. In order to perform structure mapping, CUPID creates schema trees and uses a tree matching algorithm to find matches. The tree matching algorithm is based on heuristics that are rather intuitive. For example, two non-leaf elements of their respective schema trees match if they are linguistically similar and their sub-trees are similar.

The primary focus of GLUE, PROMPT and CUPID is in identifying semantic correspondences. That is, these tools answers the question "*Which* concepts in two ontologies map to each other?" which only addresses the matching problem. The task of 'mapping' two concepts deals with the generation of "a query to transform an instance of one concept to that of the other" . These concepts may contain equivalent information, but their representations can be inconsistent. Therefore, the question "*How* do concepts in two ontologies map to each other?" is not addressed in this work. Still, the task of automated ontology matching is an important one and the value of this work is quite significant. In order to define a mapping a set of matching entities must first be defined. CUPID and PROMPT can be leveraged to automate this task. As has been noted, the reliance of GLUE on instances annuls its applicability to performing matches between federate simulation schemas. That being said, these three algorithms show how one can take advantage of structure and existing relationships to define new ones. This is a

fundamental knowledge reuse concept that can be employed in the development of an algorithm to generate transformations between federate representations of shared simulation entities.

The key points noted in reviewing existing work in the field of schema and ontology management are summarized as follows:

- A relationship between two entities can be captured using an intermediate mapping structure and a set of morphisms to translate data to and from the mapping structure. Given is advantages (stated above) the ontology-based framework should take on this approach to map simulation entities.

- As explained in the OI model, relationships between disparate simulation ontologies can be captured by including all simulation ontologies into a larger federation domain. Within this domain, relationships can be instantiated between the included sub-domains.

- Existing algorithms that are meant to relate schemas and ontologies focus on matching, not mapping. Still, these algorithms can be leveraged to support the simulation concept mapping process, given that in order to instantiate a mapping, a match is prerequisite.

- Reuse/ Inference methods employed by PROMPT and CUPID to determine new matches based on existing ones (and existing relationships in the ontology) can be leveraged to perform mapping inference.

2.5 Chapter Closure

In this chapter, existing work in the domains of simulation integration and schema management have been reviewed. At the outset, the goal of this review was to identify relevant work that can be leveraged in the development of (i) a framework for ontology-based capture and relation of simulation concepts and (ii) an algorithm to automate the aforementioned relationship definition. In the context of the former, the HLA framework for representing simulation concepts and integrating federate simulations was investigated. The HLA OMT was identified as a meta-model that could be leveraged for the specification of a vocabulary for representing simulation concepts in an information model. The AFF and BOM frameworks were identified as two methods by which the process of developing a federated simulation can be simplified or automated. Here, we learned that use of meta-data is key is supporting reuse and automation in federation development. In Section 2.2, the limited functionality of commercial simulation based design packages with respect to supporting simulation integration was discussed. ModelCenter's click-and-connect environment was identified to be an intuitive user-interface for the framework proposed in Chapter 1. Finally, frameworks for schema management were explored in Section 2.3. Here, the key components for representing relationships between concepts were identified and algorithms to automate schema matching were presented. These algorithms themselves can be leveraged directly to

support automated matching between coupled concepts in a federation. Moreover, their key features such as the idea of structure-based inference of matches can be extended to automate the definition of mappings between simulation concepts in an ontology. Table 2.2 summarizes the limitations of existing work that will be addressed in this thesis and key ideas and development that can be leveraged to do so.

Table 2.2: Summary of Key Points and Limitations Discussed in Literature Survey

Framework	Limitations and Key Points
HLA	**Key Points to Leverage:** ▪ The HLA OMT can be leveraged to development a metamodel for capturing simulation concepts in an ontology. This is one model for representing simulation domains; others include those presented in Section 2.3. ▪ To facilitate a greater degree of automation, a framework for developing simulation federations should be able to integrate federate simulations 'as-is'. One approach to do so is outlined in the AFF. ▪ The BOM framework uses meta-data to reuse piece parts in SOM and FOM development. **Limitations:** ▪ The HLA FEDEP prescribes the modification of federate simulation code to be consistent with the FOM. ▪ The generation of converters in the AFF is not automatic. ▪ The BOM piece-part approach to federation development is not viable for supporting automation in federated simulation outside of HLA

Table 2.2 (continued)

Schema Management Models	**Key Points to Leverage:** ▪ A relationship between two entities can be captured using an intermediate mapping entity and a set of morphisms to translate data to and from the mapping structure. ▪ The mapping composition operation can be leveraged to define relationships between simulation concepts automatically by chaining together an existing set of relationships. ▪ Relationships across disparate (simulation) ontologies can be instantiated by including a copy of the ontologies in a larger federation domain **Limitations:** ▪ The set of relationships that can be captured in an OI model is not very expressive. There is no data structure to capture transformations across related entities.
Schema Matching Algorithms	**Key Points to Leverage:** ▪ Automated schema matching tools can be leveraged as-is to find matches between concepts in participating federate simulation ontologies ▪ The structure based inference of matches employed in CUPID and PROMPT can be extended to find new mappings based on structure and vicinity to existing mappings. **Limitations:** ▪ Schema matching algorithms only determine which concepts map. Representational inconsistencies are not reported or handled.

Having completed this survey, a more clear understanding of the important features and possible approaches to address the simulation integration (specifically representational inconsistency) problem has been developed. In the next chapter, the lessons learned in undertaking this related work assessment are applied in the development of an ontology-based framework to support the development of simulation federations. The requirements of the framework, a metamodel for capturing and relating simulation concepts and the process of developing an inter-related federation are discussed in the following chapter.

CHAPTER 3

AN ONTOLOGY-BASED FRAMEWORK TO SUPPORT FEDERATED SIMULATION DEVELOPMENT

In this chapter, we describe a framework for supporting the process of establishing representational compatibility in a federated simulation. The goal of this chapter is to develop the hypotheses we have proposed at the outset of the thesis. In Chapter 1, we elaborated a vision for using ontologies to help mitigate the cost of achieving representational compatibility among a set of federate simulations. In the hypotheses, the use of ontologies to capture the semantics or 'meaning' of simulation model concepts is proposed. Here, the realization of the proposed vision is presented. Specifically, a metamodel for the representation of simulation concepts in ontologies is developed (hypothesis 2). The capture of federate simulation domains and the relationships between them, using this metamodel, is elaborated as well. Using these semantically rich ontologies, the derivation of transformation stubs to convert information between federate representations (hypothesis 3) is discussed.

3.1 Framework Components and Process Model

In Chapter 1, we have proposed an ontology-based framework to support achieving interoperability between federate simulations, with the following functional elements: (i) capture knowledge about simulation models in ontologies and (ii) apply this knowledge to derive transformations between federate representations of shared simulation concepts. In this section, we introduce the individual components that make up this framework. Together, these components can be used to realize the above-stated functional aspects of the framework. A model describing the overall process in which these constructs are used to achieve representational compatibility between federate simulations (in an automated fashion) is also elaborated below.

The components of this framework employed for knowledge capture are the *Simulation* (federate) *Ontologies* (SONT), a target *Federation Ontology* (FONT) and a meta-model for specifying these, called the *World Ontology*. The World Ontology contains metadata and specifies the structure of simulation objects, their attributes, interactions between objects (events) and data types. It also includes data structures to capture the relationship between these entities. Finally, this ontology includes a set of primitive data types and defines the relationships between them. The World Ontology expresses a discrete, abstract layer of semantics that is used to describe the concepts in individual simulation models. In other words, it defines a communal vocabulary for the specification of entities in SONTs and FONTs. By expressing all simulation concepts in terms of this vocabulary, the ‘meaning’ of each SONT entity is unambiguous, and the relationship between such

entities can be derived in an automated fashion. The structure of entities defined in the world ontology i.e. the properties of objects, events and their attributes is leveraged from the HLA OMT (IEEE 2000), which has been developed so as to be comprehensive and extensible in its ability to express simulation concepts. More details on the development of the World Ontology are specified in Section 3.3.

The SONT specifies the object and event architecture corresponding to a given federate simulation model. Based on the structure provided in the World Ontology, SONTs are specified by domain experts who play a major role in the development of a given simulation model. This process is analogous to documenting a SOM in current HLA practice. However, a SONT captures concepts and relationships between them in a formal, computer-sensible fashion that is much richer than the unstructured text that comprises a SOM. Unlike a SOM, a SONT contains knowledge that a computer can use to make inferences. A detailed explanation of SONT specification is elaborated in Section 3.4.

It has been mentioned that in order to facilitate consistent information transfer between federates in a distributed simulation, a common representation for all shared simulation entities must be defined. As has been noted in Chapter 2, this is the approach taken in the development of HLA federations (in the development of FOMs) and the theoretical model for schema management (Bernstein 2003) (in the creation of intermediate mapping objects). Similarly, in this framework, a common information model for consistent information transfer is defined in the FONT. The FONT specifies a federation-level

representation for all objects and interactions that are shared among different federates, and captures the relationships between the federate and common representations. FONT generation is explained in Section 3.6.

The above mentioned knowledge capture components of the framework are illustrated in Figure 3.1. It should be noted that the knowledge captured in all these ontologies is represented in terms of a frame-based knowledge model that is employed by numerous knowledge-based systems (Minsky 1975). This knowledge model is elaborated in Section 3.2.

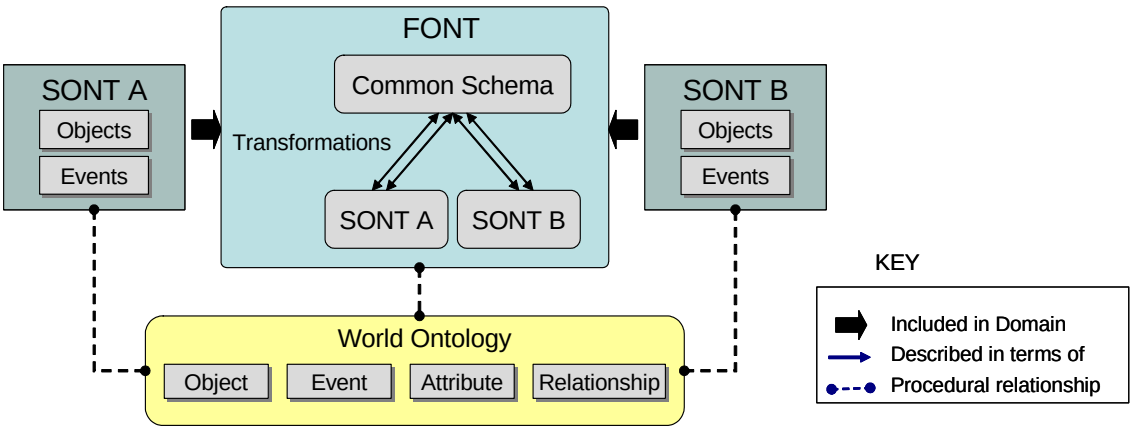


Figure 3.1: Knowledge Capture Components of Ontology-Based Framework

The second functional aspect of this framework is the application of knowledge captured in the ontologies to derive transformation stubs for converting information between federate and common representations. This is achieved through the development and application of an algorithm that queries the ontologies to infer transformations between

simulation entities. The **Gr**aph-based **I**nference of **T**ransformations (GRIT) algorithm employs existing work in graph theory and traversal to determine (i) the common representation of shared concepts in a federation (i.e. the contents of a FONT) and (ii) the transformations between federate and common representations of shared concepts, in an automated fashion. The development and functioning of the GRIT algorithm is detailed in Chapter 4.

Having introduced the individual components in the framework, let us visit the process in which these components contribute to support the process of achieving interoperability. The ontology-based federation development process can be summarized in the following steps:

- Define the World Ontology (one-time task)
- Define SONTs based on the World Ontology
- Generate a FONT
    - Determine a common information model
    - Generate the transformation routines

The overall federation development process model is illustrated in Figure 3.2. Essentially, the World Ontology metamodel needs to be defined before any other tasks can be carried out. However, the definition of this ontology is a one-time task—once created, the same World Ontology is reused over and over. In order to do so, the World Ontology must be 'included' in every SONT being developed. The inclusion of one ontology in another

means that the Domain of Discourse (DoD) of the including ontology spans at least the domain of the included ontology. In other words, every concept defined in the included ontology is also defined in the including ontology. In this manner, every SONT contains the same set of metamodel structures and relationships as defined in the world ontology.

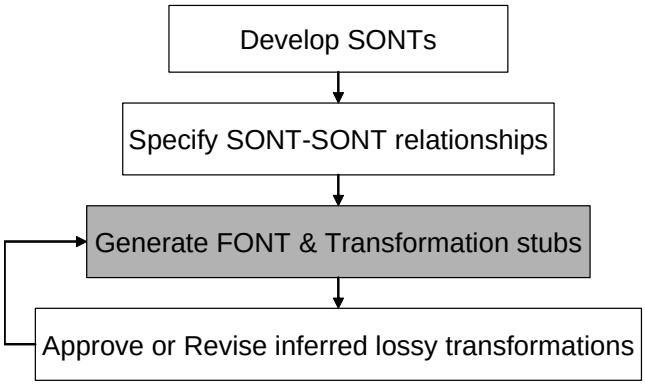


Figure 3.2: Overall Process Model for Integrating Federate Simulations in the Ontology-based Framework (box in grey indicates automated task)

Individual SONTs are specified by simulation model developers. The concepts in a given federate simulation model are to be captured in terms of the metamodel structures specified in the World Ontology. This includes simulation objects and events, their attributes and complex data types (other than the primitive set defined in the World Ontology).

Once the SONTs for the complete set of federate simulations in a distributed simulation have been specified, a common, federation-level representation for all shared entities can be defined in the FONT. In order to generate a common federation-level schema, the set

of related federate concepts must first be specified. The federation developer(s) must specify the set of participating SONTs and indicate which set of federate objects and events are related (ultimately, all relationships are defined between federate and common representations of shared entities). In order to capture a relationship between two federate entities, both those entities must be part of the same domain. Therefore, each SONT that is participating in the federation must be included in the FONT. Equivalently, the FONT must span all SONT DoD's (illustrated in Figure 3.1). Once this is accomplished, relationships between two or more objects or events can be captured in the FONT. Since objects and events are defined in terms of their attributes, so too must the relationships between them. Therefore, the federation developer must indicate which attributes of related objects and events match. It is important to note that schema and ontology matching tools identified in Chapter 2 (PROMPT and GLUE) can be leveraged to support automation of this process. However, we focus on the development of a system to perform automated mapping of matched entities. Therefore, in the framework presented in this thesis, automated ontology matching algorithms are not leveraged, their potential use is acknowledged.

Given the relationships specified by the user, new relationships can be inferred automatically by composing existing relationships together. GRIT uses the complete set of relationships to (i) automatically determine the appropriate common schema in the FONT and (ii) automatically generate transformation stubs between federate and common representations of shared entities. This is an iterative process that incorporates feedback from the federation developer. In order to select the common representation for

a set of related concepts, GRIT may require the federation developer to provide additional knowledge regarding the loss of information in transformations between the federate representations of related entities (this is explained in detail in Section 3.6). Having completed these steps, the user is presented with the set of inferred transformations, so as to either approve them or revise them if an available direct transformation is preferable. If revisions are made, the common representation, and the associated transformations are recomputed. In this manner, the process of defining relationships in a FONT is an iterative process that employs feedback from the user to refine previously generated common representation and transformation routines.

The steps described above provide a basic explanation of the process by which federate simulations are integrated in this framework. The earlier steps are focused on the capture of knowledge in ontologies; the latter apply that knowledge to support the process of achieving representational compatibility in a federation. Having presented a general overview of the components in this framework and the process in which they are employed, a more in-depth discussion follows. In Sections 3.2 through 3.6, the structure and use of the individual components is explained in detail, except for the GRIT algorithm, which is addressed in the following chapter.

3.2 The Frame-based Knowledge Model

In this section, a frame-based knowledge model (Minsky 1975) for representing concepts and relationships between them is presented (Figure 3.3). This model defines the foundation for capturing simulation concepts in ontologies. The World Ontology

metamodel, individual SONTs and the federation-level FONT are all defined using this knowledge representation. Essentially, this model can be viewed as the metamodel for specifying of all ontologies in this framework. Frame-based knowledge models have been employed in the development of several different knowledge representation systems. In the effort to provide knowledgebase interoperability, a protocol for accessing frame-based knowledge bases has been developed, named the Open Knowledge Base Connectivity (OKBC) protocol (Chaudhri, Farquhar, Fikes et al. 1998). The OKBC defines a standard knowledge model (based upon frame-based representation) and a set of operations that can be applied to the components of that knowledge model.

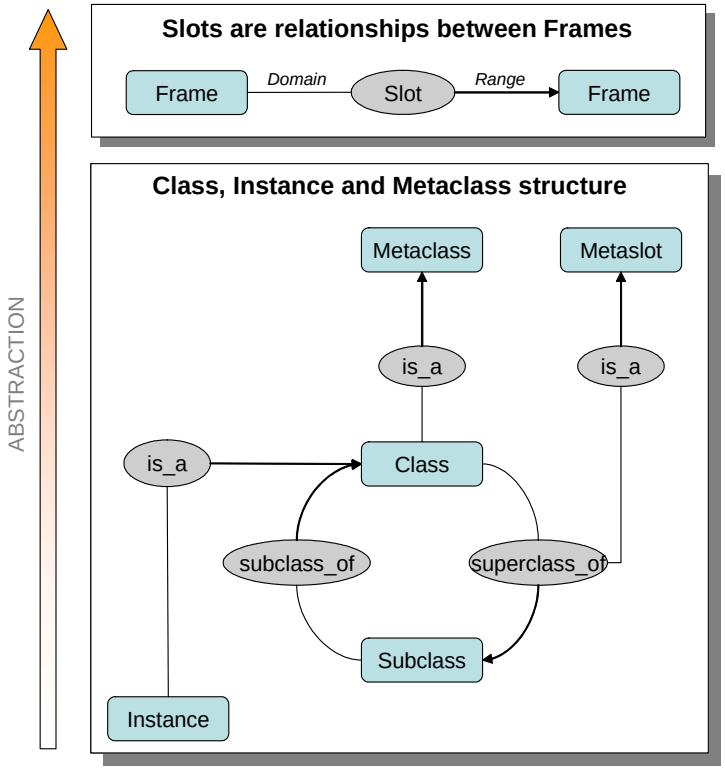


Figure 3.3: Major Concepts of the Frame-Based Knowledge Model

The knowledge model used for the development of ontologies in this framework is leveraged from the OKBC protocol. The OKBC knowledge model has been developed based on the requirements of more that fifty knowledge representation systems (Chaudhri, Farquhar, Fikes et al. 1998). OKBC was designed with the goal of being precise, flexible, extensible and consistent. As a result, the OKBC knowledge model is unambiguous and applicable to and compatible with a variety of knowledge representation systems. Furthermore, the operations defined in this protocol yield semantically equivalent results over a range of knowledge representation systems. In adopting the OKBC knowledge model as a foundation for the specification of knowledge in simulation ontologies, its above-mentioned qualities are bequeathed to this framework. This means that the framework presented in this research is based on sound, widely-accepted foundations and can be implemented, extended and interfaced with existing knowledge-based systems in a standard fashion. Parenthetically, the Protégé ontology development tool, in which this framework has been implemented, employs a frame-based knowledge model based on that which is defined in the OKBC protocol (Noy, Fergerson and Musen 2000). The OKBC knowledge model is formally defined using the Knowledge Interchange Format (KIF), which is a first-order predicate logic language (Genesereth 1995).

The elementary concept in the knowledge model employed in this framework is a *Frame*, which is a primitive object that represents an entity in a given domain. Relationships between frames are captured through the definition of *Slots*. Slots can be viewed as attributes of frames: each slot has a domain i.e. the frame to which it applies (the frame it

describes), and a range (also called value-type) i.e. the frame(s) that represent the values the slot can take. Formally, a slot is a binary relation, and each value V of a slot S of a frame F represents the assertion that the relation S holds for the entity represented by F and the entity represented by V. As an example, consider that concept of a parent and a child are captured in two frames. The relationship between the *Parent* and *Child* frames can be captured in two slots, *has_child* and *has_parent*. The *has_child* slot is in the domain of the *Parent* frame, while its value type or range is an instance of the *Child* frame (this can be intuitively read as *Parent has_child <instance_of> Child*). Slots have other properties such as cardinality (number of values a slot can have) and inverse (indicating that the slot is part of a pair that describes a reflexive relationship). Each slot can have a set of *Facets* associated with it, which represent a constraint that must hold on the relationship between two frames. Together, Frame-Slot pairs and associated facets capture the entire semantics of a DoD.

The generic frames are segregated into two object oriented constructs: *Classes* and *Instances* (or individuals). A class is an object that defines a collection of entities having the same set of properties (slots). One can view the class object as a way to represent a simulation concept in the DoD. Classes can be arranged in hierarchies with multiple inheritance. A taxonomy of simulation concepts is captured as a tree of classes, using subsumption relationships defined as *subclass_of* /*superclass_of* reflexive slots. Instances represent individual entities that are members of the class. An instance is a frame that inherits its structure (set of slots) from its defining class. The relationship between a class and its instances is captured via *is_a* slots.

Classes can be considered to be models for specifying instances. Similarly, a model for specifying classes is required: a metamodel (or a model of a model). OKBC defines a class that is the 'class of all classes' to capture this concept. In the knowledge model employed in this framework, a slightly different approach is taken. The concept of a *Metaclass* and a *Metaslot*, leveraged from the Protégé knowledge model, is used to capture collections of classes and slots, respectively. Metaclasses are frames that can be used to specify specializations of classes. Similarly, Metaslots are templates for the specification of slots with a different set of properties. These meta-modeling constructs are important for specifying the world ontology.

The overall knowledge model defined for use in this framework is illustrated in Figure 3.3. Note that this knowledge model is explained here to a degree that is sufficient to convey the development of various ontologies that comprise this framework. A more in-depth and formal specification of the concepts discussed above can be found in (Chaudhri, Farquhar, Fikes, Karp and Rice 1998).

3.3 The 'Word' Ontology Specification

Having elaborated the knowledge model used to define simulation ontologies, the application of that model to define the World Ontology is discussed in this section. As mentioned earlier, the World Ontology is meant to capture a set of metamodel concepts that can be used to define simulation model entities in SONTs and FONTs.

Concepts in the World Ontology are modeled using the Metaclass and Metaslot entities defined in the frame-based knowledge model, as illustrated in Figure 3.4. Templates for simulation objects, events and data types are defined as metaclasses, while the template for defining attributes is specified as a metaslot. This is done so that simulation objects, events and data types can be modeled as classes, each with their own set of attributes (slots that are instances of the attribute metaslot). Hence, SONT developers can take on an object oriented approach to capturing a simulation model domain in an ontology. This enables SONT developers to specify a hierarchy of objects and events, thus capturing the equivalent information specified in HLA object and interaction class structure tables.

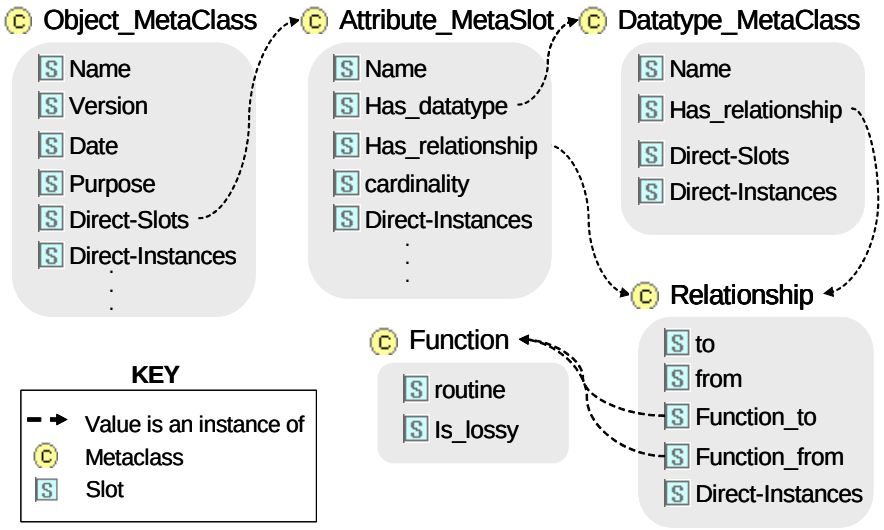


Figure 3.4: World Ontology Metamodel Components

The metaclass for objects includes slots that correspond to all fields specified in the HLA Object Model Identification Table (IEEE 2000). This includes data structures to capture

the name, version, purpose of the object and the date it was created. Furthermore, slots to capture subclass and super class relationships, class-attributes relationships (*Direct-Slots*) and aggregation relationships between the object metaclass and its instances (*Direct-Instances*) are included. The range of each slot is defined so as to constrain the definition of objects. For example, the range of the *Direct-Slots* slot is defined to be an instance of the attribute metaslot. This means that every simulation object is modeled as a class that can have one or more direct slots, as long as those slots are attributes (instances of the attribute metaslot). A detailed view of the object metaclass, its various slots and their ranges is illustrated in Table 3.1. The event and data type metaclasses are defined to be essentially identical to the object metaclass. However they are modeled as separate metamodel entities so as to be able to distinguish between object, event and data type classes in SONTs and FONTs.

The attribute metaslot has its own set of slots that define the semantics of simulation attributes, which correspond to the HLA attribute table fields. As defined in the frame-based knowledge-model, all slots must have a domain and a value-type (or range). If attributes are to be modeled as slots, the same must be true for them. Therefore, the attribute metaslot includes the slots *Has-Domain* and *Has-Datatype*. The *Has-Domain* slot is constrained such that its range is one or more instances of the object, event or data type metaclasses. In other words, an attribute must belong to an object, event or data type. The *Has-Datatype* slot is constrained such that its range is an instance of the data type metaclass. That is, each attribute can have a value type or range that is an instance of the data type metaclass. Also, a slot to capture the cardinality of an attribute (how many

values the attribute can have at a given point in time) and the *Direct-Instances* slot to capture the relationship between individual attributes and this metaslot is included.

Table 3.1: List of Slots that Define the Object Metaclass and their Value Types

Domain	Slot	Range (value is instance of….)
Object Metaclass	Name	String (data type class)
	Version	Floating Point (data type class)
	Date Created	Date MDY (data type class)
	Purpose	String (data type class)
	Sub class	Object Metaclass
	Super class	Object Metaclass
	Direct Instances	Class
	Direct Slots	Attribute Metaclass
	Direct Constraints	Facet

The World Ontology also includes a *Relationship* class to hold information about the relationship between simulation concepts in a FONT. The relationship (transformation) between a simulation entity and its common representation is represented as an instance of this class. A corresponding Has-Relationship slot is included in the definition of the meta-entities, to associate a relationship instance with a set of simulation entities. Even though the relationship class and associated Has-Relationship slots are part of the World Ontology, we postpone the discussion of the structure and use of these constructs until Section 3.5, which is devoted solely to the representation of relationships in a FONT.

Apart from template components, the World Ontology also includes a set of data types that are expected to be used consistently across all SONTs and FONTs. These include primitive data types such as integers, floating point numbers, strings, enumerated types and units of measurement. Furthermore, relationships (and transformations) between these data types can also be captured so as to be reused in myriad FONTs. However, as is explained in Section 3.5, it is more efficient to encode the relationships between these basic data types within the GRIT algorithm, than to capture them within the World Ontology.

3.4 Simulation Ontology (SONT) Creation

Simulation Ontologies are information models that represent the set of concepts defined within a corresponding simulation model. Analogous to an HLA SOM, a SONT documents the representation of the objects and events in a given simulation domain. However, in a SONT this documentation is captured in a semantically rich manner that can be processed by a software agent. The semantics describing a simulation domain are captured in terms of the metamodel concepts defined in the World Ontology.

For every SONT to be defined in terms of the same vocabulary, the World Ontology must be included in each SONT domain. The first step to be undertaken in the development of a SONT is therefore to copy the concepts defined in the World Ontology into the SONT. Having done so, the SONT developer can begin to document simulation entities as instances of the appropriate meta-entities. Simulation domains are described in an object oriented fashion in a SONT, where concepts are modeled as classes, defined by a set of

member attributes. All concepts that are viewed to be persistent throughout the length of a simulation experiment are modeled as object classes—instances of the object metaclass. Conversely, those concepts that are not persistent, i.e. they are only relevant at a single instant in time, are modeled as event classes. Each object and event must be given a unique name (name is used as a handle to identify individual frames) and are described in terms of their attributes. Given that classes (frames) are defined in terms of slots, the individual attributes of objects and events are modeled as slots; instances of the attribute metaslot. When an attribute is created within the definition of an object or event, the attribute's domain is updated to include that object or event. Note that an attribute can have multiple domains. For example, the attribute length can be a descriptor of multiple objects in a simulation model that deals with form and geometry. Each attribute must also be assigned a range; which can be one or more instances of the data type metaclass.

SONT developers may define data types for attributes as instances of the data type metaclass. Each data type can have multiple attributes that represent the individual members (fields) of that data type. Since each attribute in a data type class must have a range, which itself is an instance of the data type metaclass, data types are defined in terms of existing data types. Therefore, at some level of abstraction, all SONT specific data types are defined based on the primitive set of data types defined in the World Ontology.

Once instantiated, the object, event and data type classes can be arranged in hierarchies. This arrangement should be defined in terms of the subsumption relationships between

various entities. An object A is a sub-class of an object B if the set of attributes in A subsume that of B. In other words, subclasses inherit the attributes of their super class. The arrangement of SONT entities in hierarchies is an important step in the development of a federated simulation. When a relationship between two entities is identified, that relationship can then be applied to infer additional relationships between the respective sub and super classes of those two entities. Furthermore, during the execution of a federated simulation, if federate entity A determines its state (attribute values) by subscribing to entity B in another federate, then the former is notified of all changes to the subordinate classes of B. Therefore it is vital to arrange SONT entities into hierarchies to support relationship inference and inheritance during run-time interplay.

An illustration showing the definition and arrangement of SONT entities in terms of World Ontology concepts is provided in Figure 3.5. Note that no new relationships (instances of the relationship class in the World Ontology) are defined in the development of a SONT. That task arises when individual SONTs are to be integrated in a given federation. The representation and instantiation of relationships between SONT entities is addressed in the following section.

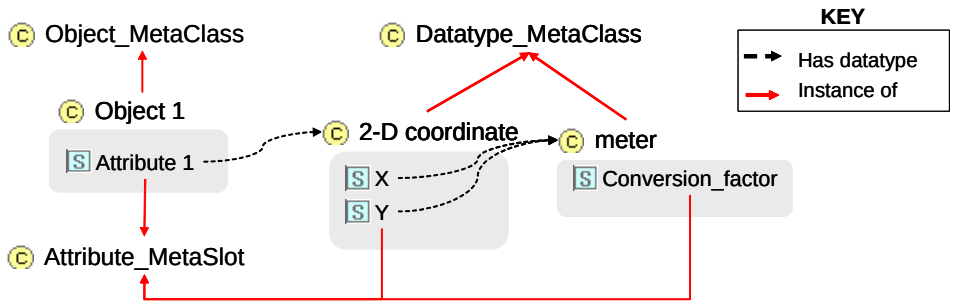


Figure 3.5: SONT Specification in terms of World Ontology Concepts

3.5 Capturing Relationships between Ontologies

The capture of relationships between federate representations of shared concepts in a federation is a key aspect of achieving representational compatibility in this framework. In this section, the semantics of relationships between different SONT entities in a federation are explored. During the development of a federation, a human must indicate a set of federate SONTs and the relationships between the individual entities they describe. Once specified, these relationships should be applied to infer transformations, and should be reused when the federation is modified. Therefore, it is important to capture these relationships in a formal manner. As indicated earlier, the user specified relationships and inferred transformations are to be stored in the FONT. In order to do so, a vocabulary for describing relationships must be specified at the meta-model level. Such a vocabulary should capture a match between two entities (i.e., the fact that two representations of a concept are related) and a corresponding mapping between them (i.e., how to transform an instance of one attribute to that of another). The definition of this vocabulary is elaborated below:

3.5.1 The Semantics and Instantiation of Relationships in a FONT

The World Ontology includes a *Relationship* class to hold information about the relationship between attributes, objects and events in a FONT. The relationship between a particular entity and its common representation is represented as an instance of this class. To account for both the match and mapping aspects, the *Relationship* class consists of the following slots:

- *to*: whose value is the target entity,
- *from*: whose value is the subject entity,
- *function_to:* whose value is an instance of the *function* class and holds information about the transformation from the subject entity to the specified target,
- *function_from:* which is analogous to *function_to*, except going from the target entity to the subject

The *function* class consists of two slots: *Routine* and *Is_lossy* (Figure 3.4). The *Routine* contains the transformation stub to convert entity instances between their federate and common representations. *Is_lossy* has a Boolean value that indicates whether the transformation from one representation to the other leads to a loss of information. In Section 3.6, we discuss the use of *is_lossy* to determine the common representation for a set of related SONT entities.

Federation developers can specify relationships between objects, events, data types and their attributes in a FONT as instances of this relationship class. In most cases, users are to specify knowledge of matches—the existence of a relationship between two SONT objects, events or data types in the federation. Based on the class-level matches, users must specify matches between the individual attributes of matched entities. Matches are specified by providing values for the *to* and *from* slots of a given relationship instance. Using this knowledge and the semantics of each matched entity the GRIT algorithm is employed to determine the routines of the *function_to* and *function_from* slots in an

automated fashion. In a minority of cases, users may explicitly provide these routines as well. To associate a given relationship with a set of SONT entities, each object, event, data type and attribute includes in its definition, a *has_relationship* slot. The range of this slot is an instance of the relationship class. In essence, the *has_relationship* slot captures the knowledge as to the participation of a simulation entity in a particular relationship.

In the FONT, all user-defined relationships are defined directly between SONT entities. Once a common representation for all shared entities has been defined, all relationships are to be defined between the SONT and common representations of entities (Figure 3.6). In doing so, the relationship between *n* SONT entities is captured as a set of independent relationships between each SONT entity and the common representation. The independence of these relationships means that when the representation of a given SONT entity changes, only the relationship between that entity and the common representation is affected. Furthermore, the instantiation of a relationship between a given SONT entity and a set of *n* already related SONT entities involves the specification of only one additional relationship (as opposed to *n*). In this manner, the capture of relationships to and from the common representation of shared entities helps to simplify the modification and reuse of a FONT.

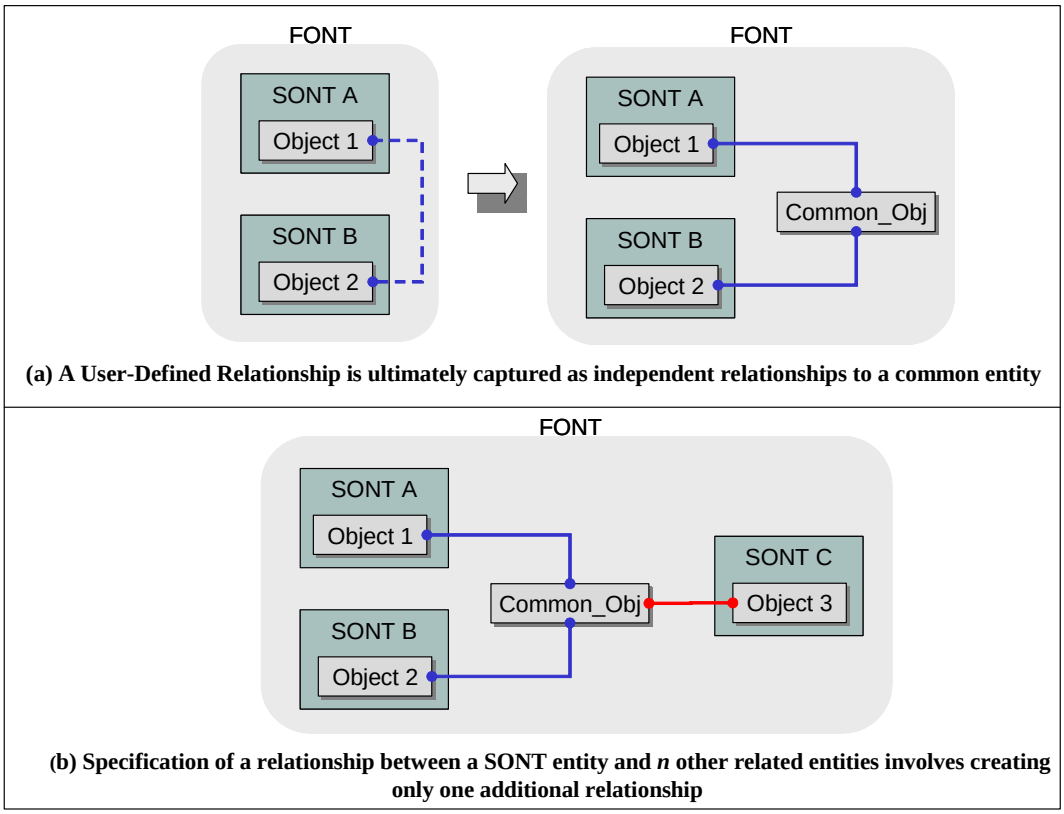


Figure 3.6: Specification of Independent Relationships to and from a Common Representation

Given that each object and event is described in terms of its attributes, the relationship between two objects (or events) is defined in terms of the relationship between their attributes. Each attribute may be matched and mapped to its common schema equivalent. A mapping between two objects or events is simply the collection of the mappings between their individual attributes. Consider the following example: two objects, *Person* and *Individual*, exist. *Person* is described in terms of the attributes *name* and *age*. Similarly *Individual* is described in terms of attributes *given name* and *years old.* A mapping from *Individual* to *Person* involves a transformation of an *Individual's given name* and *years old* attributes to the *name* and *age* of a *Person*, respectively. Assuming

that these attribute level transformations already exist, there is no need to explicitly specify a mapping between *Individual* and *Person*; the equivalent knowledge exists in relationships between their attributes. Therefore, at the class level, mappings do not need to be captured. However, it is necessary to capture knowledge as to which objects and events relate to each other to facilitate information exchange at run-time. This is because an attribute can be part of more than one object or event. In the example above, consider that the *age* attribute is part of a third object, *Tree*. While the concepts of *Person* and *Individual* relate to each other, they do not relate to *Tree*. When the *years old* attribute of an *Individual* changes; that change is to be reflected in the *age* of a *Person*, but not that of a *Tree*. This knowledge is captured in a match between *Person* and *Individual* (and the lack of a match between *Person* and *Tree*). In general, when an attribute that is part of multiple SONT objects changes its state in the context of one particular object, only those objects that subscribe to that need to be notified as to the change in that attribute's state. The same routine is used to convert that attribute's value from its SONT to common representation, irrespective of the context (domain) in which that attribute is modified. The knowledge as to which corresponding attribute to reflect these modifications in is captured in object (and event) level relationships.

Once the user specifies all SONT-SONT relationships, additional instances of the relationship class are automatically created to capture the match and mapping between SONT and common representation of entities. Essentially, these relationships capture equivalent knowledge as the relationships instantiated by the federation developer. However, it is important to maintain both sets of relationships in the FONT. Saving the

user-defined relationships in the FONT allows users to revisit and modify these relationships at any time. The automatically generated SONT-Common relationship instances capture equivalent knowledge in a form that can be directly employed by the RTI. Furthermore, by capturing relationships to and from a common representation, the task of modifying or specifying additional relationships is simplified, as mentioned above.

3.5.2 Defining Relationships between Multiple Entities

It is important to note that based on the semantics of the relationship class, a relationship can only be specified *between a pair of entities*. That is, the cardinality of the *to* and *from* slots of the relationship class are constrained to be one. The reason for this constraint relates to the generation of transformation procedures corresponding to a relationship. For every relationship instance for which a mapping is to be defined, there must be a procedure to transform information from the representation specified in the *from* slot to that which is specified in the *to* slot, and vice-versa. These procedures can be written in any choice of object oriented programming language (OOPL), such as Java or C. In the majority of OOPL's, procedures or functions are constrained to have only one return value. Therefore, any transformation stored in a relationship instance can only output a single information construct. Since relationships are bi-directional and two transformations are generated, both the *to* and *from* slot values are constrained to hold a single entity. Obviously, this limits the set of relationships that can be instantiated in a FONT. It is likely that one SONT may represent a simulation concept using several attributes whereas another may model the equivalent with a single attribute (probably of a

more complex data type). In general there may be a need to relate *n* attributes of an object in one SONT to *m* attributes of a coupled object in another SONT. The same may be true at the class level (although less likely) as well. It is important to provide for the specification of such relationships in this framework.

Several approaches can be adopted to incorporate the specification of transformations between multiple sets of entities. One such approach would be to pass arrays or lists of simulation entities as input and output arguments to the transformation routines in a relationship. However this requires additional information to be specified as to the order of the entities contained within the above-mentioned list. A more elegant solution exists: For a set of *n* attributes being related simultaneously, a new data type class—and aggregation—is instantiated such that its member attributes correspond to the set of attributes being simultaneously related. Further, a new attribute is specified such that its value type is the new data type specified above. In this manner, the information expressed in *n* attributes is now encapsulated within a single attribute (Figure 3.7). The relationship between *n* attributes in one SONT and *m* attributes in another can equivalently be expressed as a relationship between two (complex) attributes.

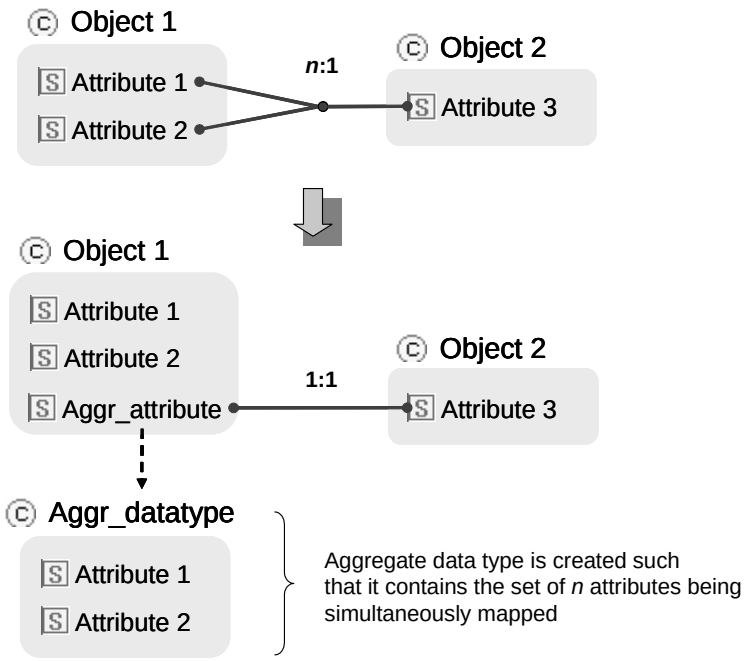


Figure 3.7: Definition of n:1 Relationships by Aggregation

As an exemplification of this approach, consider the following scenario: A person, described in terms of his or her name, is modeled using two attributes, *First Name* and *Last Name* in one SONT, while the equivalent concept is modeled in a single attribute *Full Name* in another SONT. A FONT is to be generated in which the attributes *First Name* and *Last Name* together relate to the attribute *Full Name*. Following the approach stated above, a new data type *Aggregate_Name_Type* is created by the federation developer, with the former attributes as its slots. Next, a new attribute is *Aggregate_Name* is instantiated, of value type *Aggretate_Name_Type*, within the definition of the person object. Finally, a relationship instance is specified between *Full Name* and *Aggregate_Name*.

Having aggregated a set of attributes in the SONT there is still the issue that the underlying simulation model employs a representation wherein the attributes are not aggregated. When information is exchanged by the RTI at run-time, an aggregated set of attributes must be de-aggregated when passed to the federate simulator. To address this issue, a given instance of the aggregate data type can be created such that it references the same location in memory as the original set of *n* attribute values of an object, event or data type instance. In effect, this eliminates the need for performing a de-aggregation operation. When the state of the aggregate attribute is updated, so too are the states of the attributes it encapsulates.

3.5.3 Data type relationships

One final type of relationship that merits attention is that of relationships between data types. The relationship between two attributes involves a relationship between their respective data types. Hence, it is important to capture relationships and associated transformations between data types in a FONT as well. Data type classes are very similar to objects and events. Therefore, the relationship between two data types can be captured in a similar fashion i.e. in terms of the relationship between their constituent attributes. A key difference in the representation of relationships between data types and objects/ events is that transformations are captured at the class level in a data type relationship. That is, the mapping between the constituent attributes of two data type classes are encapsulated in a single procedure (this helps to simplify the process in which attribute-level mappings are generated, as shall be explained in Section 3.6.2).

While all relationships involving user-defined data types are captured via instances of the relationship class, it is redundant to do so for the set of primitive data types (such as the relationship between two units of length measurement—meter and centimeter) defined in the World Ontology. These relationships are to be defined once and for all, when the World Ontology is developed. Rather than specifying individual relationships between primitive data types, the knowledge as to the transformations between primitive data types is encoded as part of the GRIT algorithm. Since these relationships hold true for any federation, there is no downfall in 'hard-coding' these relationships. As an example, consider the relationships between data types corresponding to different units of measurement. The relationship between two units of a certain measurable quantity is of multiplication or division by a constant conversion factor. A certain system of measurement can be chosen as a reference to which all conversion factors are determined. (1995) has shown that with the knowledge of the conversion factors relating a set of simple units (e.g., meter, second and Kelvin), the conversion factor for any composite unit (i.e., a product or quotient of simple units, such as meter per second) can be derived. Therefore, rather than instantiating multiple relationships between individual units of measurement in the World Ontology, these relationships are derived as required based on the knowledge of conversion factors. Hence, a *conversion_factor* slot (as shown in Figure 3.5) is included in the definition of simple unit data types and composite units are captured as a product of simple units. SONT developers may specify additional unit data types in the same form; the transformation between two units of a certain measurable quantity will be automatically generated based on the specified conversion factor.

We conclude this section with a quick recapitulation of the points made above. To begin with, a metamodel for capturing relationships between SONT entities in a FONT was presented so as to capture both matched and mappings between entities. Furthermore, we established that for object and event level relationships, only the knowledge as to the matches between objects and events need be specified. We saw that *n:m* relationships can be specified by the process of attribute aggregation. Finally, the capture of relationships between data types was discussed. Having done so, the reader has a fundamental understanding of how relationships between simulation entities are captured in this framework. In the next section, the process of FONT generation, of which relationship definition is a key aspect, is explored in detail.

3.6 Federation Ontology (FONT) Generation

A federation ontology (FONT) serves as a common model to and from which federates can convert shared information during run-time. Therefore the FONT consists of (its own representation of) all shared objects, events and their constituent attributes in a given federation. Further, this ontology must include the definition of the relationships between the SONT and common representations of shared concepts. In order to specify a relationship between two entities, both entities must be defined in the same ontology. Therefore, as stated earlier, the FONT includes all SONTs, a common schema that is a liaison between individual SONT representations of shared concepts and a set of relationships between them (Figure 3.1).

The FONT generation process, a sub-set of the overall framework process model, is presented in Figure 3.8. The first step in FONT generation is creating a new ontology that includes all the SONTs that are part of the federation. This simple task is analogous to including the World Ontology in each SONT. Following this, the federation developers must specify the knowledge as to which SONT objects relate to (publish or subscribe to) each other. In the previous section, the process of specifying these relationships has already been elaborated upon. When a relationship between two or more SONT entities is instantiated, a *common or shared* representation for those entities must be created. Ultimately, all relationships must be defined between federate and common representations of shared concepts.

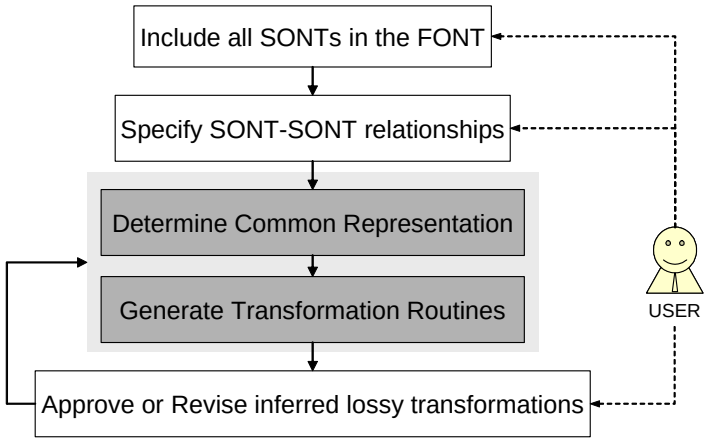


Figure 3.8: FONT Development Process Flow (grey boxes indicate automated tasks)

Given the relationships specified by the user, new relationships can be inferred automatically by composing existing relationships together. The complete set of

relationships is used to determine the appropriate common representation (Section 3.6.1). Following this, the procedural transformations associated with these inferred relationships are also composed automatically (Section 3.6.2). During these steps, the user is prompted to provide additional knowledge about transformations as required. Having completed these steps, the user is presented with the set of inferred relationships and transformations, so as to either approve them or revise them if an available direct relationship is preferable. If revisions are made, the common representation, and the associated transformations are recomputed. In this manner, the process of defining relationships in a FONT is an iterative process that employs feedback from the user to refine automatically generated common representation and transformation routines.

3.6.1 Determining the Common Schema

Since the relationship between SONT entities is captured in the FONT as a relationship between each entity and a corresponding common representation, the question arises: which representation should be chosen as the common representation? For each set of related SONT objects, a corresponding common object is specified, such that its attributes comprise the common representation of the individual SONT attributes of the objects that are related to each other. In this manner, a common schema of objects, events and their constituent common attributes is defined in the FONT. Alongside, a set of relationships between SONT entities and their common schema equivalents is instantiated.

There is still the question as to what the representation of a common attribute should be. By making the common attribute correspond directly to one of the SONT attribute representations, at least one of the transformation routines will be trivial. It is furthermore important to choose a representation that avoids any unnecessary loss of information when exchanging data in a federation. The importance of this choice may not be evident when there are only two related attributes; in fact it is irrelevant in this case. However, this choice becomes significant when three or more SONT attributes in a federation relate to each other. For example, if the SONT attribute *position* of data type *2-D coordinate* relates to attribute *location* (in another SONT domain) of type *3-D coordinate*, and attribute *point* also of type *3-D coordinate*, the corresponding common attribute must be of type *3-D coordinate*. If it is selected to be of type *2-D coordinate*, then there is an avoidable and unrecoverable loss of information. Both attributes *location* and *point* have three coordinates, yet when *location* subscribes to *point* (or vice-versa), the value is converted from *3-D* to *2-D* (common representation) and back to *3-D*, resulting in a loss of the third coordinate's value. To avoid this scenario, the common representation of a set of related attributes *should* have a representation that does not lead to any avoidable loss of information.

In order to determine which SONT representation of a shared attribute is the appropriate common representation, we introduce the notion of *lossiness*. A transformation from one representation to another is *lossy* if any information is lost in that transformation. In the example above, the transformation from attribute *location* to *position* is lossy (while the inverse is not). The information about lossiness is captured in the *is_lossy* Boolean slot of

a given instance of the *function* class. In a relationship where *from* = *position* and *to*= *location*, the value of *function_to* (an instance of the function class) has *is_lossy* = true, while that of *function_from* has *is_lossy* = false.

The common representation for a set of related attributes is best determined as the representation that leads to the *fewest* number of lossy transformations. In the event that there are several SONT representations that lead to the same (least) number of lossy transformations, any of them may be picked as the common representation. In the case of the example presented above, it is clear that if the representation of the SONT attributes location or point is selected to be the common representation, then the number of lossy transformations is at its minimum (Figure 3.9).

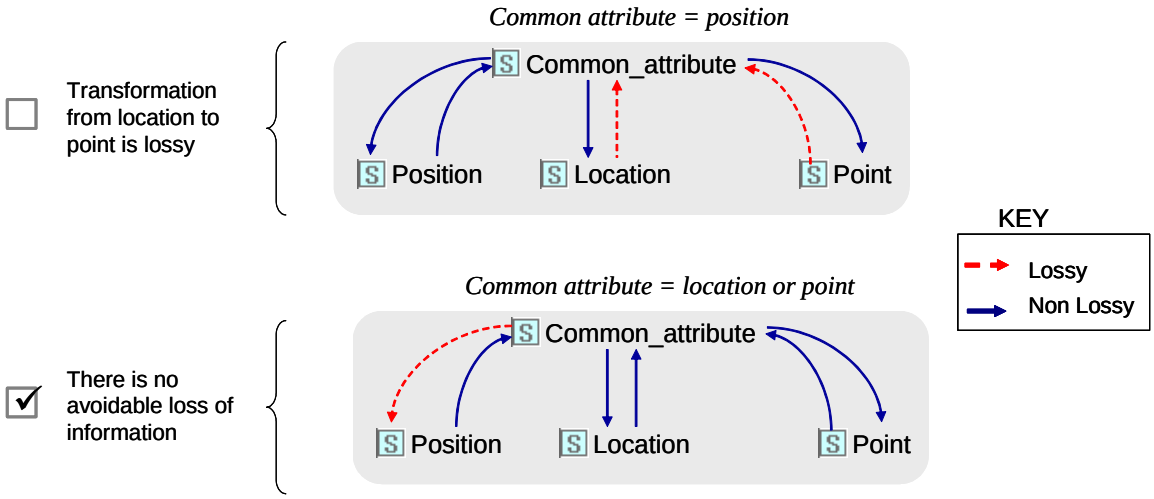


Figure 3.9: Selecting an Attribute Representation Resulting in the Smallest Number of Lossy Transformations

Generally, knowledge as to the lossiness of transformations between related attributes is provided by the federation developer. While the selection and instantiation of the common schema is automated, the knowledge required to do so must be explicitly provided by a human(s). This information is conveyed by providing a value (*True* or *False*) for the *is_lossy* slot in a given relationship's functions. Based on the knowledge provided by the user, the GRIT algorithm simply selects a common representation that leads to the smallest number of lossy SONT to SONT transformations. Once the knowledge of lossiness in a given mapping is provided, it can be applied to determine lossiness of other transformations. Specifically, if the mapping between the data types of two related attributes involves a lossy transformation, then the corresponding attribute-level transformation must be lossy as well. However, the fact that there is no information loss in the mapping between two attribute data types is not sufficient to determine if the attribute-level transformation is lossy. It is important to understand that there can be a loss of information in a transformation between two attributes having the same data type. For example, a transformation from the attribute *Name* to *First Name* involves discarding information about a person's surname, even though both attributes have a value type of *String*. Essentially, lossiness relates to the underlying semantics of the concepts captured in related attributes than to their data types.

The complexity of the task of determining the common representation for a set of related attributes varies based on the set of user defined relationships. This point is highlighted using the example depicted in Figure 3.9. Consider that the federation developer specifies a relationship between the *position* and *location* attributes and another between *location*

and *point*. In order to determine which of these attributes should be selected as the common representation, the GRIT algorithm notes the number of lossy transformations associated with each of them being (hypothetically) selected as the common representation. To determine the number of lossy transformations associated with *point* being the common representation, knowledge as to the lossiness of a transformation from *position* to *point* is required. This knowledge is not specified in the set of user defined relationships, and potentially the federation developer may not know how *position* and *point* relate. Therefore, the relationship between these two attributes is derived by composing existing relationships. A transformation from *position* to *point* can be inferred transitively as the sum of the transformation from *position* to *location* and that of *location* to *point* (Figure 3.10). Based on the lossiness of these transformations, the required knowledge of lossiness in the transformation from *position* to *point* is determined. Once a count of lossy transformations associated with each federate attribute is obtained, a suitable common representation can be selected.

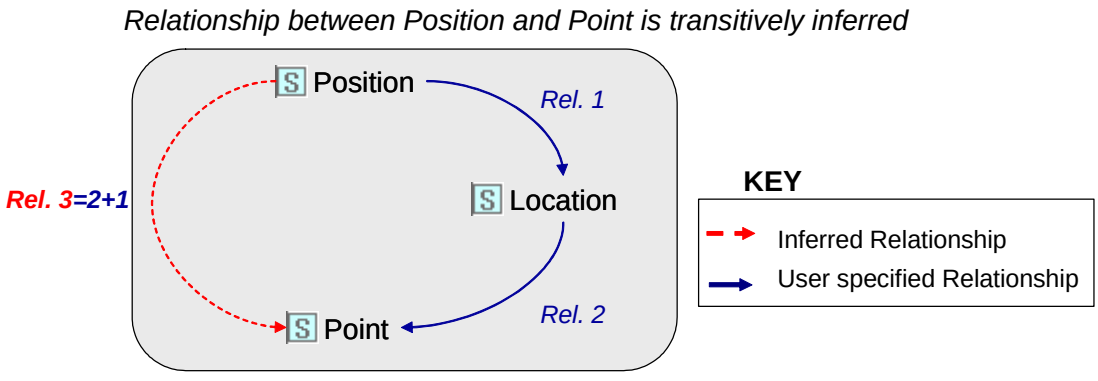


Figure 3.10: An Example of Inferring Relationships by Composition

Having determined the common representation for a set of related SONT attributes, a corresponding instance of the attribute metaslot is instantiated in the FONT. Furthermore, additional instances of the relationship class are created to capture the match between each SONT attribute and the newly instantiated common attribute. These steps can be performed automatically and do not require any interaction from the federation developer. When a common representation and corresponding relationships have been defined for the complete set of related attributes of two SONT objects (or events), a common representation of those objects is also automatically instantiated, such that the member attributes of this common object are the common attributes defined earlier. As was done with related attributes, a new set of relationship instances are created to capture the match between the SONT and common objects.

Having created a common set of entities, a final step in the development of the common schema is to arrange these entities into a hierarchy. This step is vital to facilitate inheritance in publication and subscription of federate objects or interactions. That is, if a certain object subscribes to another SONT's parent object, it should be notified of all updates to the children of that parent object. The set of common objects and events in the FONT are arranged into a hierarchy based on Classification—the process of constructing a concept hierarchy in which more general concepts are located above more specific ones according to the subsumption order. The subsumption relationship between two objects in a schema is defined such that an object B *subsumes* an object A if the set of attributes that comprise B includes the set of attributes that comprise A. In this case, object B is a refinement of object A, or A is the parent of B. The hierarchical arrangement of common

objects and events is captured be specifying ranges for their respective *subclass-of* and *superclass-of* slots.

At this point, a common representation for all shared entities is defined and ordered, and a set relationships between SONT and common entities is instantiated. In this section, we explored the process by which these tasks are undertaken, except for the implementation of the GRIT algorithm to select the common representation. A significant aspect of the following chapter is devoted to studying the development and application of the GRIT algorithm. The final step in the development of the FONT is the generation of transformation stubs to convert SONT entities to their common representations and vice-versa. Specifically, the *function_to* and *function_from* slot values in attribute level-relationships need to be defined. The generation of transformation stubs in an automated fashion is discussed in the following section.

3.6.2 Generating Transformation Routines

The transformation routines in a relationship between two SONT entities comprise a mapping between those two entities i.e. they specify a procedure by which an instance of one SONT entity can be converted to an instance of the other SONT entity and vice-versa. The generation of transformation routines in this framework is analogous to the specification of converters in the AFF (Macannuco, Dufault and Ingraham 1998) and schema morphisms in the framework for model management developed at Microsoft (Alagic and Bernstein 2001). While the specification of converters and morphisms is a manual process, we exploit the use of ontologies to infer transformations in an automated

fashion. As mentioned earlier, mappings only need to be defined for attribute relationships. Since all classes are defined in terms of their attributes, the mapping between two simulation objects, for example, can be viewed as a collection of mappings between their attributes. The process by which attribute-level transformations are generated is detailed in the following paragraphs.

There are two discrete conversions that are encapsulated in an attribute-level transformation stub: (i) a conversion between the data types of two related attributes and (ii) a conversion between the two concepts being related. The data type conversion deals with the fact that two simulation entities that model the same concept may have different representations. The data type conversion is employed to convert an instance of a simulation concept from one federate representation to another. The second conversion deals with the fact that two distinct concepts may be related in a federation. Apart from the fact that these concepts may have different representations, there is a fundamental relationship between the concepts themselves that needs to be captured as a procedure. As an example, consider that a federation developer specifies a relationship between the attribute *radius* of type *meter* in one SONT and *diameter* of type *foot* in another. Clearly, these two concepts are related, but they are not the same. Furthermore, they are expressed in dissimilar length units. The transformation between the SONT attribute radius to diameter consists of two discrete conversions—one to covert the concepts of radius to that of a diameter and the other to convert the resulting value in meters to feet. The resultant transformation stub is as follows:

```
Function_to.routine:
foot radius_to_diameter (meter radius) {
   foot diameter;
   diameter= (meter_to_foot(radius))*2;
   return diameter;}
```

It is important to note that this framework does not constrain the types of conceptual relationships that can be defined between simulation entities. In most cases, relationships in a federation are defined between two representations of the same simulation concept. Here, the concept level conversion is simply one of equivalence, and the only non-trivial conversion is between the data types of two attributes. In cases where relationships are defined between disparate concepts, the federation developer must specify the knowledge as to how these two concepts relate. Ideally, we would like the user to specify this relationship in a declarative fashion, from which the transformations in either direction can be derived (e.g. `radius – (diameter/2) = 0`). However, a declarative relationship between two entities can be converted into two procedures (to perform transformations in either direction) only if that relationship is analytically invertible. Hence we assume that whenever the user explicitly specifies a mapping, he or she does so in a procedural form (e.g. `radius=diameter/2 and diameter =radius*2`).

The extent to which a transformation stub can be generated autonomously depends on the conceptual relationship between two attributes, and the relationship between their data types. The relationship scenarios and the corresponding steps undertaken to arrive at the required transformation stubs are listed in the following cases:

Case 1: The conceptual relationship between two attributes and the relationship between their data types is specified by the user. Most relationships across SONTs are made across entities that refer to the same concept. If the conceptual relationship between two attributes is one of equivalence, no additional knowledge needs to be specified in the relationship definition (apart from knowledge of a match). In contrast, if the mapping between two attributes involves a relationship between two distinct concepts, the required transformation must be specified explicitly by the user.

Given that a relationship between the data types of a pair of related attributes is known, the required attribute-level transformations can be inferred automatically. In a majority of cases, attributes are defined in terms of primitive data types. Recall that the relationship between primitive data types is already is already known (as discussed in Section 3.5.3). Therefore, transformations between primitive data types can be instantiated, which are then used to perform the attribute-level transformations. For example the transformation from data type *meter* to data type *foot* can be generated automatically based on the knowledge of the conversion factor captured in the World Ontology:

```
foot meter_to_foot (meter input) {
   foot output;
   output=(input/foot.conversion_factor
   *meter.conversion_factor);
   return output;}
```

If both data types are not primitive, then the transformation between them requires additional knowledge. When the federation developer specifies a relationship between

two attributes such that one or both of them has a custom data type, a relationship must be defined between those to data type classes. To do so, the user must specify a set of matches between the individual attributes of these data types. Based on this, the data type level transformation can be generated automatically as a collection of the transformations of its individual attributes. Consider the example transformation between the *position* and *location* attributes illustrated in Figure 3.9. Position is defined to have data type *2-D coordinate* (with attributes *x*, *y* and *z* of data type *foot*) whereas location is of type *3-D coordinate* (consisting of *x* and *y* in *meters*). The relationship between *3-D coordinate* and *2-D coordinate* can be derived automatically if the user specifies that the respective *x* and *y* fields equate to each other. Since these fields have primitive data types, the following transformation function between the custom data types is generated automatically:

```
3D 2D_to_3D (2D input) {
   3D output;
   output.x = meter_to_foot(input.x);
   output.y = meter_to_foot(input.y);
   output.z= 0;                // user specified default
   return output; }
```

Data type level transformations can be applied to generate the required attribute-level transformation stub to convert *position* to *location* or vice-versa. This routine is created as follows, assuming that *position* and *location* refer to the same concept:

```
3D position_to_location (2D input) {
3D output;
output= 2D_to_3D(input);
return output;}
```

Case 2: The relationship between two attributes is not explicitly defined by the federation developer. When the common representation for a set of related attributes is defined, a set of relationships between the SONT attributes and the selected common representation are instantiated. Depending on the selection of the common representation, some of the SONT-common relationships have to be composed from the set of user defined relationships. The transformations corresponding to these composed relationships cannot be derived as elaborated in the case 1. If the user has not defined a given attribute relationship explicitly, he or she has not captured the knowledge as to the conceptual and data type conversions between those attributes. Moreover, the federation developers may not know anything about the relationship between two attributes that they did not explicitly specify.

To determine the lossiness in relationships that are not defined by the user, the approach of composing user-defined relationships together is taken on, as explained in Section 3.6.1. A similar approach is taken to generate transformation stubs corresponding to composed relationships. We use the *position-location-point* transformation example developed thus far to illustrate this approach. Consider that the user specifies relationships between *position* & *location*, and *location* & *point*. Upon the selection of *point* as the common representation, a transformation to map *position* to *point* is required. This transformation is represented as a nested procedure in which calls are made to

transformations from *position* to *location* and *location* to *point*, as illustrated below. Obviously, the latter transformations are either explicitly specified by the federation developer or are generated automatically as elaborated in Case 1.

```
3D Position_to_Point (2D input) {
   3D output;
   output = Location_to_Point(Position_to_Location(input));
   return output; }
```

Transformations between data types for which relationships have not been specified are also composed in the same manner as explained above. It should be noted that a composed transformation will involve a loss of information if any of its constituent transformations are lossy. The GRIT algorithm is devised so as to search for the smallest chain of transformations that does not involve a lossy transformation. However, in the event that there are no non-lossy chains, the composed transformation will inevitably lead to a loss of information (Figure 3.11), which may be potentially avoidable. When the FONT has been completely defined, the federation developer is presented with the set of composed relationships with lossy transformations. If the user is cognizant of a direct relationship and an associated non-lossy transformation to replace a composed lossy transformation, he or she may explicitly define this relationship and provide the knowledge required to generate the required transformations. The common schema and transformation generation process is then reiterated with the revised set of relationships. In this manner, the framework incorporates user feedback in the development of the FONT.

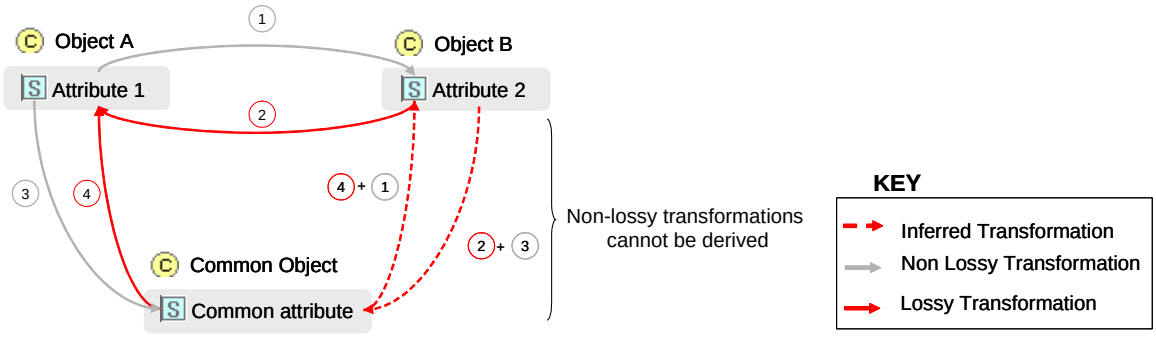


Figure 3.11: Deriving Transformations by Composing Existing Ones

Once the final common schema and transformation routines have been approved, the FONT generation process is complete. At this point, a federation level representation for shared concepts is defined, and all run-time simulation interplay is conveyed in this format. Also, a set of transformation stubs are generated for the RTI to convert information it sends and receives from federate simulators into the appropriate representations. Given that all this knowledge is stored in the federation ontology, the procedure by which an RTI accesses and interprets this knowledge is an important issue that remains outstanding. However, the development of such an RTI-FONT interface is outside the scope of this framework and is hence not discussed. Suffice it to mention that several web-based markup languages for ontology serialization exist, such as Resource Description Framework (RDF) and Web Ontology Language (OWL) (Davies, Fensel and Van Harmelen 2003), which along with their respective parsers, can be leveraged in the development of an RTI-FONT interface.

3.7 Assessing the Structural Validity of the Framework

Having explored the integration of federate simulations using the ontology based framework, the reader's attention is now focused on assessing the structural validity of this framework. As mentioned earlier, the internal consistency of this framework is accepted by ensuring that it is based on sound foundations. In the previous chapter, the salient points of existing work that should be leveraged in the development of this framework were identified. Here, we briefly recapitulate to demonstrate that the development of this framework draws from these key points.

First, the specification of the World Ontology to represent simulation concepts is based on the definition of the HLA OMT (IEEE 2000). The set of simulation entities (objects, events, attribute) and their various properties (value type, cardinality etc.) was ascertained based on the tables and fields that comprise the OMT. Furthermore, the approach to integrating simulations through the specification of a federation-level FONT is leveraged from the FOM based federation development approach in HLA (Defense Modeling and Simulation Office (DMSO) 1999). Finally, the principle of inferring new relationships based on existing ones, identified in the PROMPT (Noy and Musen 2000) and CUPID (Madhavan, Bernstein and Rahm 2001) algorithms to perform ontology matching, is leveraged to support automation in generation of mappings between matched simulation entities. A similar approach has been demonstrated in the AFF, where the importance of being able to chain exiting converters together is highlighted .

The overall framework process model is also developed by building upon the work of others. Specifically, the process of relating simulation entities via an intermediate mapping entity and a set of transformations is derived from the schema management models developed by (Alagic and Bernstein 2001). The approach of instantiating converters between federate and federation-level representations has also been employed in the AFF, albeit specific to the HLA framework. As has been demonstrated in the AFF, the reusability of federate simulation models in multiple federations is ameliorated by taking on this approach. The same is true for the ontology-based framework—once a simulation model is developed, it can be used in multiple federations by specifying relationships and transformations between its entities (captured in the SONT) and the common model defined for that federation. In this manner, the need to modify the simulation model each time it participates in a new federation is diminished.

Having developed the individual components of this framework and the overall process model by building upon the work of others, this framework can be viewed to be internally consistent, and theoretically valid in its structure. However, the validity of the GRIT algorithm, which is part of this framework, is not yet accepted. The development of this algorithm and its structural validity are discussed in the next chapter.

CHAPTER 4

AN ALGORITHM FOR AUTOMATED FONT DEVELOPMENT

In the previous chapter, a framework for federation development using ontologies was developed. Thus far, the capture of semantics describing a simulation federation has been focused upon. In this chapter, we present the process by which the knowledge captured in an ontology is applied to complete a given FONT specification in an automated fashion. Referring back to the hypothesis posed corresponding to research question 3 in Chapter 1; the specification of a graph-based algorithm to infer federate-common relationships and transformation stubs is detailed in the following sections. In Section 4.1, a high-level explanation of the algorithm and its constituent procedures is presented. The application of graph theory and graph algorithms to support achieving representational consistency in a federation is discussed in Section 4.2. The specification of individual procedures that together comprise the algorithm being presented is detailed in Sections 4.3 thru 4.5. Finally, this chapter is closed out with a discussion to support the acceptance of the theoretical structural validity of this algorithm.

4.1 Overview of the GRIT Algorithm

A key step in the ontology based federation development process model (Section 3.2) is the automated generation of the common schema and transformations in the FONT. Using the formally captured semantics of SONT entities and the relationships between them, a suitable common representation of these entities and a set of procedures to transform between their SONT and common representations are to be specified. In order to do so, a *G*raph-based *I*nference of *T*ransformations (GRIT) algorithm is presented. As its name suggests, the GRIT algorithm uses constructs defined in the field of graph-theory to model the set of related entities in a FONT. By building upon existing algorithms to efficiently traverse a graph, the GRIT algorithm infers a suitable common representation and derives associated transformation stubs based on the existing knowledge captured in a FONT.

The process by which the GRIT algorithm accomplishes the above-mentioned tasks is illustrated in Figure 4.1. First, connected graphs are developed corresponding to the set of related SONT objects, events, their attributes, and data types. Next, graphs of objects, events and their attributes are used to determine the set of simulation entities that share a common representation, and subsequently their common representation. Following this, a set of SONT-Common relationships and their associated transformations are derived by composing user-defined relationships together, as was discussed in Section 3.7.2. Finally, transformations between data types of different related SONT attributes are instantiated,

and the ensuing common schema and transformations are presented to the federation developer for approval or revision.

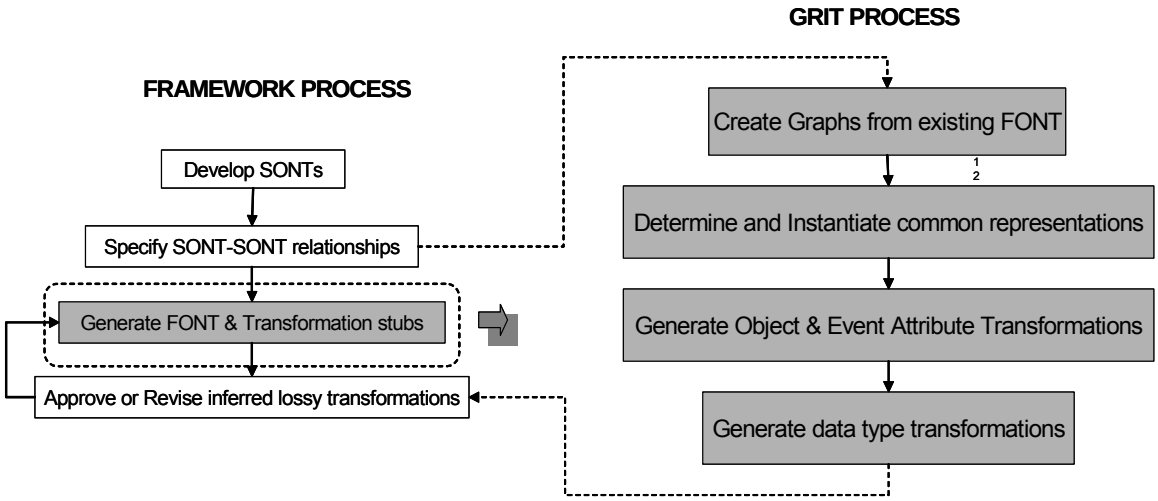


Figure 4.1: GRIT Algorithm Process Model

To demonstrate how the GRIT algorithm supports FONT generation through its different steps, we employ the *Position / Location* example developed in the previous chapter. To briefly recapitulate, consider that a SONT consists of the object *Vehicle*, which has an attribute *Position* of data type *2-D Coordinate*. Vehicle is related to the object *Car* in another SONT domain, which is described in terms of the attribute *Location*, of data type *3D-Coordinate*. There exist relationships between the attributes *Location* and *Position*, and the data types *2-D Coordinate* and *3-D Coordinate*. The GRIT algorithm is employed to (i) create a common representation for the attributes *Location* and *Position*, and the objects *Vehicle* and *Car* (ii) create transformation stubs between these SONT entities and their common representations, and (iii) create transformation stubs between the *2D and*

3D Coordinate data types. Throughout this chapter, we refer to this example to help explain how each procedure in the GRIT algorithm contributes to accomplishing these tasks.

The steps depicted in Figure 4.1 comprise a single iteration of the GRIT algorithm. As mentioned earlier, the FONT development process is iterative; hence these steps may be repeated several times before the final FONT specification is obtained. Within the GRIT algorithm, each of these steps is executed by an individual procedure. In the following sections, the specification of these individual procedures are discussed in detail. In order to study these procedures, a sound understanding of basic graph theory, the representation of a FONT as graphs and graph traversal algorithms employed in GRIT is pre-requisite. Therefore, we begin below with a basic and brief introduction to graph theory and associated algorithms.

4.2 Graph Theory and Algorithms

Graph theory is a field of mathematics that deals with the use of diagrams or graphs to study the arrangement of objects and the relationships between them. Graph theory is defined as “the study of graphs, either for their own sake, or as models for such diverse things as groups (in pure mathematics) or computer networks” (Kuperberg 2000). Indeed, the theory of graphs has been applied extensively to solve problems in myriad domains, including optimization, electrical engineering, communication & network engineering and programming (Bondy and Murthy 1981; Yellen and Gross 1998). In this thesis, graph theory is applied to support automation of the process of deriving procedural

relationships between simulation model entities in the ontology-based framework presented thus far. Our interest in graph theory is not so much in the existence of proofs for specific theorems, but rather in the application of efficient algorithms developed to perform graph-relevant tasks. Algorithms to find a path between two nodes in a connected graph are of specific relevance to the generation of a FONT. The FONT consists of a set of related entities for which a common representation and associated transformations are to be generated in an automated fashion. The user-specified knowledge in a FONT can be used to construct a connected, directed graph of related simulation entities. Using this graph, an algorithm to find the shortest path between two vertices can be employed. This shortest path is essentially equivalent to the most trivial transformation from one simulation entity (vertex) to another. In this chapter, the implementation of this proposed graph-based approach to infer the common schema and transformations is detailed. Before doing so, we introduce the basic concepts in graph theory and the graph algorithm being used in this thesis.

A Graph G (V, E) is a structure that consists of a set of *Vertices* V= $\{V_1, V_{2...}\}$ and a set of *Edges* E= $\{E_1, E_2\ldots\}$. Each edge is incident on a pair of vertices (which are not necessarily distinct), thus connecting them. A graph is qualified to be *directed* if its edges have an ordered pair of end points (vertices), such that and edge E (u, v) starts at u and ends at v. Associated with each edge is a length (or weight), which is non-negative. The length of an edge usually defines some characteristic of the connection between two vertices, such as its complexity, or time required to travel along that edge. A path within a graph refers to a sequence of edges such that E_I and E_{I+1} share a common end point.

Obviously, in a directed graph, the end vertex of E_I in a path is equivalent to the start vertex of E_{I+1}. A graph that is not fully connected is called a *forest*. Each forest may have several connected sub-graphs and trees (which are sub-graphs without circuitous connections). The concepts of a forest, sub-graph, vertex, edge and path are all illustrated in Figure 4.2. A more in-depth discussion on the concepts in graph theory can be found in (Even 1979; Gross and Yellen 2003).

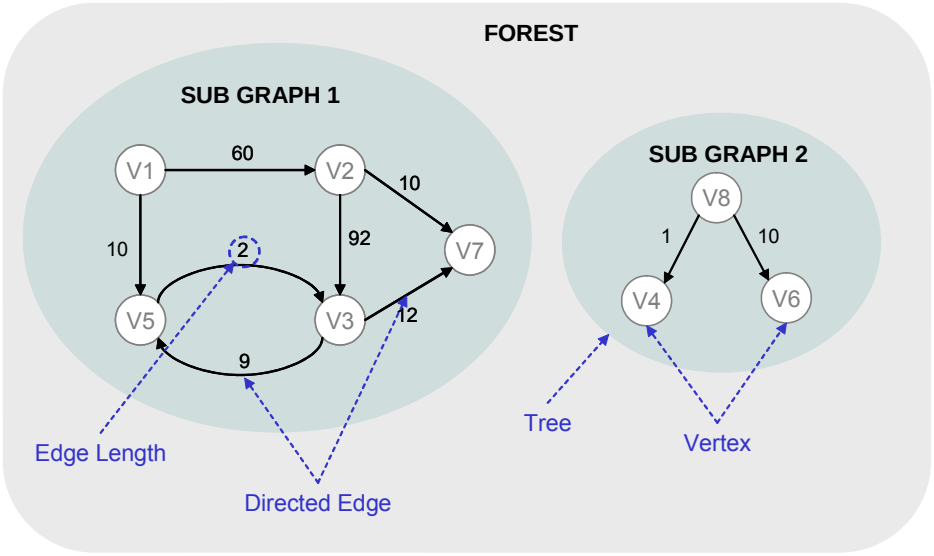


Figure 4.2: Basic Concepts that Comprise a Directed Graph

Several efficient algorithms have been developed to perform operations on graphs, such as graph traversal and tree ordering. As mentioned above, the algorithms of particular interest are those that identify paths between two given vertices. An algorithm to determine the shortest path between two vertices is used to support the selection of a common representation and associated transformations in the FONT development process. As we shall see in following sections, the edges that constitute the shortest path

between two vertices in a graph is equivalent to the chain of functions composed together in a transformation from one simulation entity to another.

The two famous, widely-accepted and used shortest-path algorithms are Floyd's algorithm (Floyd 1962) and Dijkstra's algorithm (Dijkstra 1959). Floyd's algorithm, also known as the all-pairs algorithm is used to find the shortest path between every pair of vertices in a graph. In contrast, Dijkstra's algorithm finds the shortest path between a specified vertex and all other reachable vertices in the graph. Essentially, Dijkstra's algorithm can be modified to execute *n* times so as to achieve the same functionality as Floyd's algorithm. Given its simplicity, we use Dijkstra's algorithm in the GRIT algorithm described in this chapter. There exist several variants to Dijkstra's algorithm in existence; we use the baseline sequential single-source algorithm, which is specified below in pseudo-code:

```
procedure Sequential Dijkstra
{
  d_s =0                                   //intialize distance of source vertex
  d_i (i ≠ s)=∞                            // initalize distance of target vertices
  T=V                                      //initalize untraversed vertices
  while T ≠ empty
  {
    find all V_m ∈ T, with minimum d_m      //i.e. the closest vertex to V_s
    for each E(V_m, V_t), V_t ∈ T           //untraversed vertices directly connected to V_m
    {
      if (d_t >[d_m +length(E(V_m, V_t))])  //i.e. a shorter path to V_t exists
          d_t =[d_m +length(E(V_m, V_t))]
    }end for
    remove V_m from T                      //update untraversed vertices
  }end while
}end procedure
```

The sequential Dijkstra algorithm is explained as follows: Consider a graph with N vertices and a set of directed edges. Given a source vertex V_S, the algorithm listed above finds the shortest distance of N-1 vertices from V_S. The distance of a given vertex V_I from V_S is represented as d_I. The set of vertices in the graph is designated as V, and the set T refers to those vertices that have been visited (the shortest distances to those vertices has been determined). Dijkstra's algorithm begins with an initialization phase, in which the distance of V_S from itself (d_S) is set to zero, and all other distances are infinite. The set of un-traversed vertices (T) is initially the entire set of vertices (V) in the graph.

The algorithm repeatedly picks a set of vertices V_M from T, having minimal d's, which guarantees that the shortest path is always explored. Initially, V_S has the minimum distance of zero (all others are infinite). Therefore, the algorithm always begins its trace from V_S. At this point, the set of vertices V_T in T that are adjacent to V_S are identified. This is done by comparing the lengths of all edges with start point V_S. If the distance of the vertices V_T from V_S is calculated to be less that what is previously recorded, the new minimal distance is noted as d_T. In the initial iteration, all recorded distances are infinite, and given that all edge lengths are finite, the distance of vertices adjacent to V_S are noted as length of the edges E (V_S, V_T). Finally, V_S is removed from the set of un-traversed vertices and the process is repeated. That is, having noted the distances of the vertices adjacent to V_S, the adjacent vertices with the smallest distance are identified as V_M, the distances of the vertices adjacent to them (V_T) are noted and so on. In this manner, the algorithm always takes incremental steps along the shortest path(s) moving away from V_S, until all reachable vertices have been traversed. Note that the distance of those

vertices that are not part of the same sub-graph as V_S i.e. are not reachable from V_S, remains infinite when the algorithm completes its execution. A simple example to highlight the execution of the sequential Dijkstra algorithm is illustrated below in Figure 4.3.

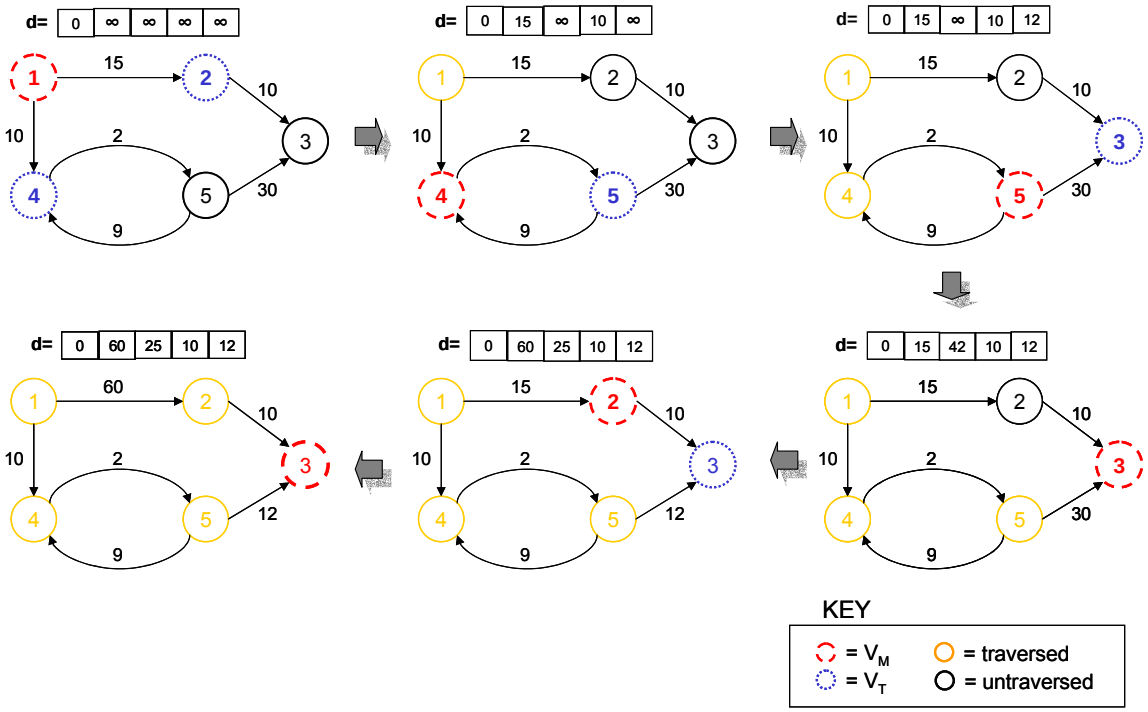


Figure 4.3: Example Execution of Sequential Dijkstra Shortest-Path Algorithm

The complexity of Dijkstra's algorithm is quantified using the Big-O notation (Ryan 1992) or Landau notation, which is a theoretical measure of the complexity of an algorithm, usually indicating time or memory required for execution. Dijkstra's algorithm is of complexity O(N^2), meaning that the maximum or upper bound on the number of operations performed or time taken to do so is $\leq (\alpha * N^2)$, where N refers to the number of

vertices in the graph. The value of α depends on the cost of performing comparisons and other operations within the nested loops.

Dijkstra's algorithm determines the shortest distance from a given vertex V_s to all other vertices in a graph. In doing so, the algorithm uncovers the shortest path from V_s to any other vertex. While the algorithm does not report this path, given the shortest distance between two vertices, the task of identifying the shortest path is trivial. A modification to this algorithm, as employed in GRIT, reports both the shortest path and distance between two vertices in a graph. It is important to note two fundamental conditions that are pre-requisite for Dijkstra's algorithm to function properly: (i) the graph should be finite and (ii) the lengths of edges must be non-negative and finite. As we shall see, both these conditions are satisfied in the application of this algorithm to compose relationships in the FONT generation process. Before exploring the application of Dijkstra's algorithm in the development of GRIT, the representation of FONT as a graph, in a convenient, computer-interpretable form is presented in the following section.

4.3 Representing a FONT as a Directed Graph

A graph is a model of a set of entities and the connections between them. This is equivalent to what is modeled in an ontology. Therefore, the application of graph theory to this framework is straightforward. A given simulation attribute in the FONT that participates in a relationship is represented as a vertex in a graph. The attribute-level relationships, as defined by the user, correspond to edges connecting two vertices. Since a relationship encapsulates two routines to transform between the related attributes, each

relationship corresponds to a pair of directed edges in a graph. Given that there are sets of attributes that relate to each other, the resultant directed graph will generally be a collection of sub-graphs, i.e., a forest.

In order to explain the GRIT algorithm for supporting automated FONT generation, it is important to first discuss how simulation entities (vertices) and their associated relationships (edges) are stored in a computer. One way to represent a graph in a computer is in an *adjacency matrix* (*A*); which is an *nxn* matrix such that A_{IJ} equals 1 if there exists and edge from V_I to V_J, or zero otherwise. For cases where edges have lengths, A_{IJ} is set to the length of the edge, or zero if the edge does not exist. It has been noted that the adjacency matrix is not an efficient representation for sparse graphs; graphs where the number of edges is small (less than n^2, where n is the number of vertices). It is likely that a graph of simulation entites and their relationships will be sparse. That is, each SONT entiity is only related to a limited set of other entities. Usually, a given SONT concept only relates to one other concept in another domain. Moreover, if relationships are specified between federate attributes A and B and B and C, it is unlikely that the user will explicitly define a relationship between A and C, as this can be inferred based on the relationships already specified. For sparse graphs, the representation of choice is that of *incidence lists*. An incident list is a list (array or linked list) of pointers to all edges incident upon a given node. Maintaining incidence lists makes tracing a path in a graph rather simple: Given a vertex, the set of edges incident upon it are immediately available (rather than having to search through an adjacency matrix).

The incidence list approach is taken to represent a simulation attribute graph. In order to capture the properties of vertices, such as the attribute they refer to and their corresponding incidence lists, an array or table of vertices is maintained. Similarly, an array of edges is constructed, holding information as to length of each edge. Each index of the vertex array is a complex data structure which includes the fields Attribute Name, Incidence List, Shortest Distance List and Shortest Path List. Given that the graph is directed, the Incidence List could potentially include pointers to both edges that start from and end at a given vertex. However, Dijkstra's algorithm (and hence the GRIT algorithm) only requires a listing of edges starting from a given vertex. Therefore, the Incidence List of a given vertex only lists the index numbers of edges leaving a given vertex. The shortest distance from the subject vertex to every vertex in the graph is captured in the Shortest Distance List. The Shortest Path List of a given vertex, as its name suggests, is an array such that its n^{th} index captures an array of edge indices, indicating the shortest path from the subject vertex to the n^{th} vertex. In other words, the shortest path list is a 2-dimensional array, or an array of arrays. Given that the number of edges in the shortest path between two edges varies for each pair of edges, each index of the shortest path list is modeled as a dynamic array. Obviously, the length of the shortest path list for each vertex is N, where N is the total number of vertices in the graph. Each index of the edge array is a complex data structure that includes the fields Start Vertex, End Vertex and Length. Start and End Vertex fields hold the index number of the start and end vertices respectively. The vertex and edge array representation of an attribute graph is depicted below in Figure 4.4.

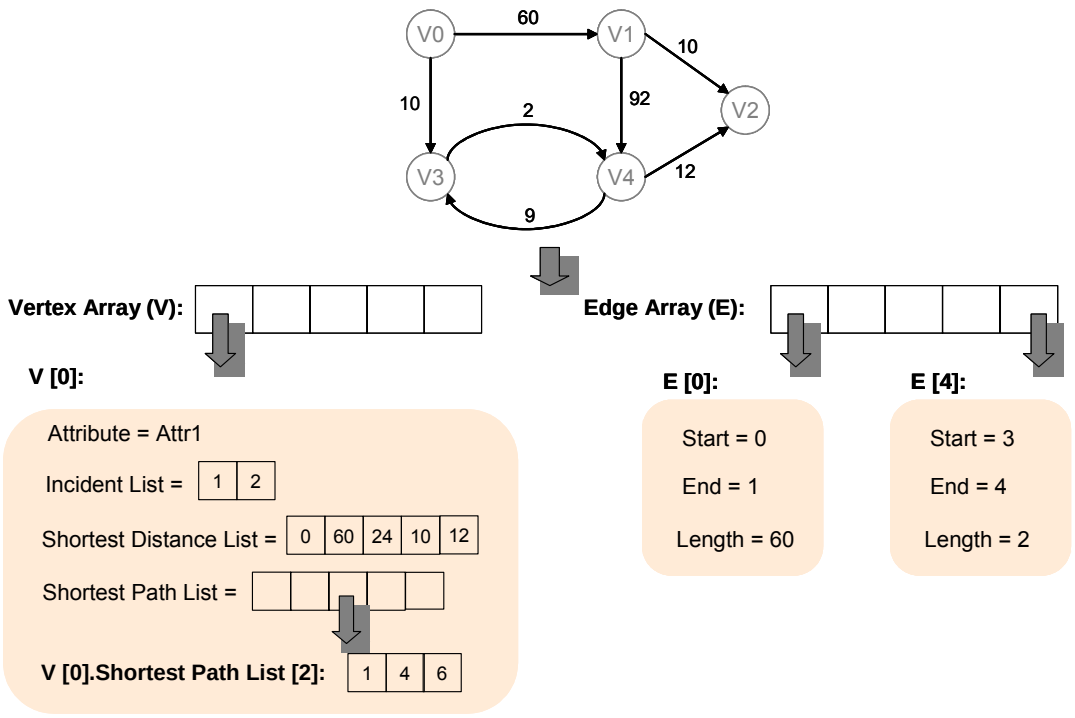


Figure 4.4: Representation of an Attribute Graph using Vertex and Edge Arrays

The length of an edge is determined based on the lossiness of its corresponding transformation, which is defined in a given attribute-level relationship. If a given transformation is non-lossy, the length of the corresponding edge is 1. Edges corresponding to lossy transformations are given the weight *2*m*, where m is the total number of edges in the graph. Assigning contrasting weights to lossy and non–lossy transformations allows the GRIT algorithm to compose the least lossy relationship between two attributes as the shortest path between their corresponding vertices. At first glance, one might question why lengths of 1 and *2*m* are selected. We need to be able to differentiate between lossy and non-lossy transformations. Since we are employing a shortest path algorithm, a non-lossy transformation should have a smaller length. Furthermore, we want to be able to distinguish between a single non-lossy

transformation, and a chain of non-lossy transformations. Similarly, it is important to discern one lossy transformation and a chain of two or more lossy transformations. Clearly, we need to use finite, non-zero lengths to accomplish the latter (choosing weights of zero or infinity means that one cannot differentiate between one edge and a chain of edges). We cannot use negative and positive weights for non-lossy and lossy transformations, respectively, because the shortest path will then correspond to the longest non-lossy transformation. Moreover, if a lossy transformation is given a positive length *L* and a non-lossy transformation corresponds to an edge of length 1, we cannot differentiate between a single lossy transformation, and a chain of *L* non-lossy transformations. However, if we choose the value of *L* to be greater than the total number of edges in the graph, we are guaranteed that the length of a single edge corresponding to a lossy transformation will always be greater than the longest possible chain of non-lossy transformations. Therefore, by selecting lossy and non-lossy edges to be of length *2*m* and 1, respectively, we avoid the case where one cannot discern between a chain of non-lossy transformations and a single lossy transformation.

Given a graph that is represented as indicated above, a modified version of Dijkstra's algorithm is used to generate the shortest path list corresponding to each vertex in that graph. This modification entails capturing the actual shortest path between two vertices, rather than just the distance of that path. The shortest path between two vertices specifies the set of edges that constitute the shortest connection between those vertices, and the order in which they are traversed. This information is vital to generate the transformation stubs in the relationship between two SONT attributes. Each edge in a path corresponds

to a routine that is nested in the composed transformation from one attribute (vertex) to another. The order in which these edges are traversed determines the order in which the corresponding routines are nested. An in-depth explanation as to generation of transformation stubs using shortest path lists follows in Section 4.5.

The procedure to generate the shortest path lists for each vertex in the graph is as follows:

```
procedure Generate Shortest Path List {
 for (i=0, i<length(V), i++){
    V[i].shortest distance list[i]=0
    V[i].shortest distance list[ j ≠ i]=∞
    T=V
    while T ≠ empty{
      find all V[m] ∈ T with minimum (V[i].shortest distance [m])
      for each E_j ∈V[m].incident list: E_j =E(V[m], V[t]), V[t] ∈ T{
         if (V[i].shortest distance list [t] >(V[i].shortest distance list[m]+E[j].length)) {
            V[i].shortest distance list [t] =(V[i].shortest distance list[m]+E[j].length)
            V[i].shortest path list[t] =append (V[i].shortest path list[m], j)
         }end if
      }end for
      remove V[m] from T
    }end while
  }end for
}end procedure
```

The application of Dijkstra's algorithm in the above listed procedure is evident. The procedure to determine the shortest path of all reachable vertices from a given start vertex (V_S) is applied *N* times (where *N* is the number of vertices in the attribute graph), so as to determine the shortest path between every possible pair of vertices. The minimum distance test from the original algorithm is applied to determine if a shorter path from V_S

to a given vertex V_T is found. In the above listed procedure, the minimum distance and shortest path to V_T are updated simultaneously. Since the shortest path is captured as an array of edge index numbers, the shortest path to a vertex V_T is captured by appending the index of the edge corresponding to the shortest path between vertex V_M and V_T to the existing shortest path between V_S and V_M. Note that for every vertex V_Z that is not reachable from a given start vertex V_S, the shortest path from V_S to V_Z is not captured and the Z^{th} index of V_S's shortest path list remains null. When the procedure ends, the shortest path list for each index of the vertex array V, i.e. the shortest path from V[i] to every other reachable vertex, in V is defined.

The complexity of this procedure is of the order $O(N^3)$. Essentially, this procedure iterates Dijkstra's algorithm *N* times, where *N* is the total number of vertices or equivalently the total number of related SONT attributes in the FONT. The cost of this algorithm can more accurately stated to be $\leq \alpha * (N^2 * m)$, where m is the number of attribute-level relationships defined in the FONT. Since it is highly unlikely that a relationship is defined between each attribute in the FONT and all other attributes, the number of edges *m* will almost always be less than *N*. Most often, the number of relationships involving a given attribute depends directly on the number of SONTs in the federation. That is, a given attribute is only related to those entities that model the equivalent concept in another domain. Therefore, assuming that the number of SONTs in a federation will be relatively small, the procedure listed above is of relatively low complexity, and will not require substantial amounts of resources (operations, time and memory) to complete its execution.

Having elaborated the representation of related FONT attributes as a directed graph and the procedures to initialize the associated arrays, we may proceed to studying the application of this graph to determine the common representation and transformation stubs for a set of related entities. At this point, it should be mentioned that two distinct attribute graphs are maintained by the GRIT algorithm. One is used to determine the common representation and transformations between the attributes of simulation objects and events. The other graph consists of the attributes of data types and the relationships between them. The reason being: data type level transformations are to be generated in an automated fashion as well; however there is no common representation defined when two SONT data types are related. To illustrate this point, we return to the *Vehicle—Car* example introduced earlier. In the FONT, the set of related attributes pairs are *Position & Location*, *Ordinate & X*, and *Abscissa & Y* (the latter two are attributes of the *2D* and *3D Coordinate* data types). For the attributes *Position & Location*, a common representation is to be defined, whose data type will be either *2D* or *3D Coordinate*. Since the common representation is defined in terms of federate data types, data types and their attributes do not have common representations. Hence this step is to be skipped for *Ordinate & X* and *Abscissa & Y*. Therefore, it is necessary to differentiate between attributes of objects (and events), and those of data types. To do so, these are maintained in separate graphs.

4.4 Common Representation Generation

The development of a common representation for a set of related entities is an important step in the development of a FONT. As has been elaborated in Chapter 3, a common attribute is to be instantiated for a set of related SONT attributes. In the case of the

Vehicle—Car example, a common representation is to be defined for the attributes *Position* and *Location*, and the objects *Vehicle* and *Car*. It is through these common representations, that information about these related entities is exchanged. The common attribute relating *Position* and *Location* is to take on either one of these federate attribute representations (i.e. its data type must be selected as either *3D or 2D Coordinate*). Specifically, the SONT representation that leads to the least number of lossy transformations is to be selected as the common representation. Using the graph representation of a related collection of SONT attributes, the common representation for a set of related attributes can be determined in an efficient, automated fashion. A procedure to do so is presented in this section. Furthermore, a common object corresponding to *Vehicle* and *Car* is to be instantiated. This common object must be defined such that the common representations of all related attributes of *Vehicle* and *Car* are in its domain (in this case, the common object has a single common attribute). A subsequent procedure to generate common objects and events, and include appropriate common attributes in their domains is also developed below.

The common representation for a set of shared attributes is selected as one of these federate representations. The SONT attribute representation which leads to the smallest number of SONT-SONT lossy transformations should be selected to be the common representation. To determine this selection, the number of lossy transformations associated with each SONT representation ‘hypothetically’ being selected as the common representation must be determined. Following this, the SONT representation leading to

the least number of lossy transformations can then be selected as the common representation. This is the general approach taken in the procedure presented below:

```
procedure Select Common Representation {
cost= new integer [length(V)]
T=V
for each V[m] ∈ T{
    T2 = All V[n] : V[m].shortest distance list [n] != ∞
    for each V[i] ∈ T2{
        for each V[t] ∈ T2{
            for each E[j] ∈ V[t].incidence list {
                cost[i] =cost[i]+V[t].shortest distance list [i] +
                        V[i].shortest distance list [E[j].end]
            }end for
        }end for
    }end for
    Find a V[k] ∈ T2 : cost[k] < cost[j] for all j ≠ k, V[j] ∈ T2
    C = create common representation(V[k], T2)
    generate attribute transformation stubs ( V[k], T2, C)
    T=T-T2
}end for
}end procedure
```

Consider that the set of attributes describing objects and events and their relationships are modeled in a graph as specified in the previous section. The first problem in identifying a common representation is to determine which set of attributes share that representation. For a given vertex V[m], the set of other vertices that share a common representation is simply those vertices that are reachable from V[m]. Since a pair of edges are instantiated for every attribute relationship, we know that if V[m] is reachable from V[n], then V[n] is reachable from V[m]. Therefore, the set of reachable vertices from V[m] and V[n] are the same. Given the representation of vertices in a vertex array, a set of attributes (vertices)

sharing the same common representation as a given vertex V[m] is defined as all V[n] for which the n^{th} index of V[m]'s shortest distance list is not infinite. In the procedure listed above, this is set is identified as T_2.

Having determined the set of vertices corresponding to attributes that share a common representation, we now come to the task of determining the common representation from amongst these attributes. As mentioned earlier, this is done by calculating the number of lossy transformations associated with each attribute being selected as the common representation. Note that the transformations to be evaluated are SONT-SONT transformations, which correspond to each existing edge connecting two any two vertices (V[n], V[m]) in T_2. Once the common representation is instantiated, all SONT-SONT transformations are carried out via the common representation. Therefore, an edge connecting V[n] to V[m] becomes a path from V[n] to the common representation to V[m]. The lossiness of these end-to-end transformations are evaluated so as to select a common representation to a minimization in information lost in SONT-SONT communication at run-time. Therefore, if a given vertex V[k] in T_2 is 'hypothetically' the common representation, then each edge V[n] to V[m] is modeled as a path from V[n] to V[k] to V[m]. The lossiness the associated SONT-SONT transformation is quantified by the length of this path.

In the procedure presented above, the total length of all paths V[n]—V[k]—V[m] corresponding to every edge E(V[n], V[m]) in T_2 is recorded as the cost of selecting V[k] to be the common representation. The process is repeated for each vertex in T_2, and the

vertex corresponding to the lowest cost is selected to be the common representation. In order to determine the cost of selecting a given vertex V[k] as common, the shortest distance from V[n] to V[m] via V[k] for each edge (V[m], V[n]) is determined and added to cost[k]. This shortest distance is simply the sum of the shortest distance from V[n] to V[k] and that from V[k] to V[m]. Note that these distances are already recorded when Dijkstra's algorithm was applied to the graph, as elaborated in Section 4.3.

A simple example of the common attribute selection procedure is illustrated in Figure 4.5. A sub graph consisting of three vertices reachable from each other (V0, V1, and V2) is identified as the set T_2. The progress of the procedure in selecting each one of these vertices as the common representation is depicted in the figure. Each edge in the sub-graph is modeled via the selected common representation, and the resultant shortest distance is noted. The sum of these distances (for all edges in the sub-graph) is the cost associated with each vertex being selected as the common representation. V1 is found to have the lowest cost and is selected as the common representation. By inspection, it is clear that when either V1 or V2 is selected to be the common representation, there is only one resultant SONT-SONT transformation that is lossy, that from V1 to V0 (this cannot be avoided based on the set of relationships defined). However, if V2 is selected as the common representation, the SONT-Common relationship between V0 and the resultant common attribute has to be inferred as a chain V0—V1—Common attribute (V2). In the case that V1 is selected to be common, no SONT-Common relationships are inferred by composition. Hence, it makes sense to select V1 to be equivalent to the common

representation, with the intent of realizing the simplest SONT-Common transformations thereafter.

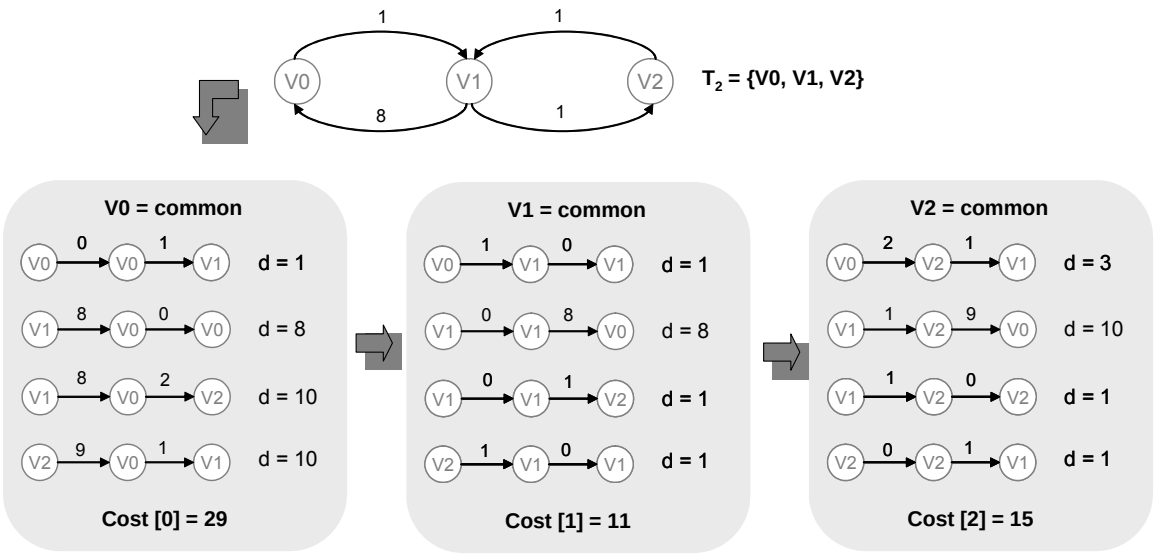


Figure 4.5: Illustration of the Graph-Based Common Attribute Representation Procedure

It is important to note that selecting the SONT representation with the lowest cost as the common representation leads to the most efficient manner in which information is converted to and from the common representation. At the outset, our goal was to select a common representation that leads to the smallest number of lossy transformations. Following the procedure discussed above, not only is a representation leading to minimal lossy transformations selected, but the composition of SONT-Common transformations (based on those specified by the user) is as simple as possible. That is, the shortest possible path between vertices (i.e. the lowest cost) implies the shortest possible chain of user-defined transformations (path of edges). There may be multiple V[k] that lead to the least number of lossy SONT-SONT transformations, but the complexity (number of

transformations composed together) of their associated SONT-Common transformations may differ. In the approach elaborated above, the representation leading to the least complex composition of transformations is always selected.

The complexity of the procedure to determine common representation is of the order $O(S*\bar{V}^2*m)$, where *S* refers to the total number of sub-graphs in the forest, $\bar{V}$ refers to the maximum number of vertices in each sub-graph and m refers to the total number of edges in each sub-graph. As has been argued earlier, the complexity of this and all graph-based procedures relates directly to the number of SONTs in the FONT and the complexity of relationships between them. Given a reasonable number of attributes (*N*) and relationships between them (*m*/2), this procedure will not require significant resources to compute the common representation for a set of related attributes.

Once the common representation for the set of related attributes in T_2 has been determined, a new instance of the attribute meta slot is created as part of the common schema. This attribute is modeled to be equivalent to the SONT attribute corresponding to the vertex in T_2 with the lowest cost. That is, its range (data type) and constraints (e.g. cardinality) are exactly the same as the selected SONT attribute. Following this, a set of relationship instances are defined between each SONT attribute corresponding to a V[i] in T_2 and the newly instantiated common representation. Furthermore, the transformation stubs for these relationships are generated, following a procedure to be presented in Section 4.5. The entire process is repeated for each sub-graph in the forest i.e. each set of

attributes that share a common representation or each set of vertices that are reachable from each other.

Having defined the common representation, a set of SONT-Common relationships and associated transformation stubs for all attributes, the common representation of objects and events and their associated SONT-Common relationships can be defined. Recall, once a common attribute is defined corresponding to *Position* and *Location*; a common object must be defined for *Vehicle* and *Car*, such that the above mentioned common attribute is in this common object's domain. The process of generating common representations for related objects and events is relatively simple, and employs a much simpler graph than is required for attributes. An undirected graph of objects (or events) is created such that each vertex in V refers to a related object, and each edge in E refers to the existence of a match between a pair of objects (vertices). The length of each edge is assumed to be constant; edge lengths are of little consequence here. Furthermore, the shortest paths between vertices in an object graph do not need to be captured. Based on this simplified graph, the object and event common representations are created by employing the following procedure:

```
procedure Generate Object/Event Common Representation {
V ∈ T
for each V[m] ∈ T {
    T2 = All V[n] : V[m].shortest distance list [n] != ∞
    if (length(T2)>1)
       C = new common object/event ( )
    for each V[i] ∈ T2{
       for each attribute aj ∈ V[i] {
          if (∃ R : R.from =aj & R.to = acj, (acj ∈ common attributes))
             add to domain (acj, C)
       }end for
    create new relationship (V[i],C)
    }end for
 T=T-T2
 }end for
}end procedure
```

Similar to the first steps of the process to generate common representations for attributes of objects and events, this procedure begins by determining the set of SONT objects (vertices) sharing the same common representation, by finding the set of vertices (T_2) reachable from a given vertex V[m]. If this set only includes one vertex, it is obvious that the corresponding SONT object or event is not related to any other objects, and no common representation is created. If a set of multiple related objects is found, a new instance (C) of the object (or event) metaclass is created as part of the common schema. This common object then needs to be described in terms of a set of common attributes. The common representations of the attributes of each SONT object in the set T_2 make up the set of attributes that describe C. As one might expect, many of the attributes of each V[i] in T_2 share the same common representation. In the procedure listed above, each attribute (A_J) of each V[i] in T_2 is queried to determine if it participates in a SONT-

Common relationship. If so, the corresponding common attribute (AC_J) is modified such that its domain includes the newly created class C (the values in the domain slot of AC_J are appended to include C). Following this, a relationship instance between each V[i] and C is instantiated to indicate a match between the SONT objects and their newly created common representation. At this point, the process of generating and relating the SONT and common objects is complete; no transformations need to be defined at the object or event level since their mappings are defined in terms of the mappings between their individual attributes. This entire process is repeated for each sub-graph in the forest of objects or events. This procedure is not of a high degree of complexity; its complexity is of the order O(S*T*a), where *S* is the number of sub-graphs in the object (or event) forest, *T* is the number of vertices in each sub-graph, and *a* is the number of attributes in the domain of each object.

Thus far, procedures have been outlined for creating common representations for objects, events and their attributes using a graph-based representation. Here we have seen that given a set of vertices, edges and the shortest distances between two given vertices, the common representation of simulation entities can be generated in an automated fashion. The remaining final component of the GRIT algorithm is the instantiation of transformation stubs between the SONT and common representation of object and event attributes and their data types. In the following section, procedures that utilize shortest path lists to generation transformations between attributes and data types, respectively, are explored.

4.5 Transformation Stub Generation

In order to map SONT attributes to their common counterparts, so as to facilitate consistent information exchange at run-time, a set of transformation stubs need to be generated as part of each SONT-Common attribute relationship. In the context of the *Vehicle—Car* example, a common representation for attributes *Position* and *Location* is generated following the procedure discussed in the previous section. Alongside, a set of relationships between these attributes and their common representation are instantiated. For these relationships, transformation stubs need to be generated. It is by invoking these SONT-Common transformation stubs that run-time information is exchanged between the corresponding federate simulations. A graph-based procedure to generate these stubs is presented in this section. Object and Event level relationships do not require transformations, as they are converted from SONT to common form (and vice-versa) by employing the transformations defined for their constituent attributes. However, a set of transformations *do* need to be defined between the related data types in the FONT (such as from *2D Coordinate* to *3D Coordinate*, and vice-versa). Since a transformation between two attributes involves a transformation between their respective data types, calls to these data type transformations are nested within attribute-level stubs. In this section, a procedure to infer data type level transformations is also presented. We begin below with the specification of attribute level transformations.

In Section 4.4, a procedure to generate a common representation for a set (T_2) of attributes corresponding to vertices that are reachable from each other was presented. Once the common representation and SONT-Common relationships are defined for the

attributes in a given sub-graph, the transformation routines in these relationships are instantiated by making a call to the following procedure:

```
procedure Generate AttributeTransformation Stubs (V[k], T2, C) {
 for each V[m] ∈ T2 {
   Find relationship Rj in FONT between V[m].attribute and C
   Create Transformation Header (Rj.function_to.routine, V[m], C)
   if (V[m] ≠ V[k]){
      for (i=0, i <length (V[m].shortest path list [k]), i++){
         Ej = E [V[m].shortest path list [k] [i]]
         Find corresponding relationship Rl : Rl is between V[Ej.start].attribute &
        V[Ej.end].attribute
         if (Rl ∈ relationships with explicitly defined tranformations) {
            Append Transformation (Rj.function_to.routine, V[Ej.start].attribute,
           V[Ej.end].attribute)
         else
            Append Transformation (Rj.function_to.routine, datatype (V[Ej.start].attribute),
           datatype (V[Ej.end].attribute) )
      }end for
   }end if
 Close Transformation (Rj.function_to.routine)
 }end for
}end procedure
```

Given a set of mutually reachable vertices (T_2), a corresponding set of related attributes, and a newly instantiated common representation (C), a SONT-Common relationship (R) *from* each attribute corresponding *to* a vertex V[m] in T_2 is identified. For every such relationship, the values of the *function_to* and *function_from* slots are to be determined. In the procedure listed above, the *function_to* routine for each R is instantiated by making calls to the Create Transformation Header and Append Transformation functions. As its

name suggests, Create Transformation Header simply sets the String value of R.*function_to*.*routine* to the appropriate function header as:

```
<C's data type> <V[m].attribute>_to_<C> (<V[m].attribute's data
type> input)
   {
     <C's data type> output;
     output = input;
```

Following this, the actual conversion from the given attribute (V[m]) to the common representation (C) is to be appended to this transformation routine. Obviously, if V[m] is the vertex V[k] who's corresponding attribute was chosen to be the common representation, the relationship between V[k] and C is one of pure equivalence, and the required transformation is simply `output = input`. Therefore, no further steps need to be appended to this transformation. However, for all other V[m], the transformation from the attribute corresponding to V[m] to C is not quite as trivial. These transformations may be composed of a chain of user-defined relationships existing in the attribute graph. A transformation from V[m] to C is equivalently traced as the shortest path from V[m] to V[k]. In the procedure listed above, the shortest path from V[m] ($V_m \neq V_k$) to V[k] is identified as the k^{th} index of V[m]'s shortest path list. For each edge E_J in this path, the equation relating *output* and *input* in the transformation being specified is modified to include a transformation from the start to end vertices of E_J. In this manner, the transformation from V[m] to V[k] (equivalently to C) is captured as a chain in which each link corresponds to the transformation associated with an edge E_J in the shortest path from V[m] to V[k].

In appending the transformation corresponding to a given E_J to the relationship between *input* and *output*, care has to be taken to differentiate between those E_J that relate two representations of an equivalent concept and those that relate two different concepts. For those E_J that correspond to relationships between two representations of the same concept, a transformation from E_J.start to E_J.end simply entails a conversion from E_J.start's data type to that of E_J.end. Therefore, the relationship between *output* and *input* in the transformation being generated is (pre) appended to include a transformation between the appropriate data types of the start and end vertices of E_J, as follows:

```
output = (<E_J's start vertex data type>_to_<E_J's end vertex
data type>(input));
```

Obviously, if the start and end vertices of E_J have the same data type, it doesn't make sense to specify a transformation between them; hence this step is skipped for all such cases.

For those E_J that correspond to relationships between two disparate concepts, a transformation between the data types of the two vertices connected by E_J does not completely describe the conversion from the start to end vertex of E_J. As mentioned in Chapter 3 (Section 3.7.2), transformations for all relationships that involve more than a data type conversion must be explicitly defined by the federation developer. Assuming that all such transformations exist and are named in the form `<attribute_1>_to_<attribute_2>`, they can be appended to the chain of transformations between V[m] and C as:

```
output = (<E_J's start vertex>_to_< E_J's end vertex>(input))
```

The transformation from V[m] to C is generated by pre-appending the transformation corresponding to each E_J, in the fashion described above, until all edges in the entire path from V[m] to V[k] have been traversed. The resultant appended transformation is of the following form:

```
output = (<Ek's start vertex>_to_< Ek's end vertex> (<E_J's
start vertex data type>_to_<E_J's end vertex data
type>(input)));
```

Having traced the entire path from V[m] to V[k], the required transformation is closed by adding a return statement. The resultant complete transformation is as follows:

```
<C's data type> <V[m].attribute>_to_<C> (<V[m].attribute's data
type> input)
   {
      <C's data type> output;
      output = (<Ek's start vertex>_to_< Ek's end vertex> (<EJ's
      start vertex data type>_to_<EJ's end vertex data
      type>(input)));
      return output;
   }
```

The transformation between each attribute corresponding to a vertex V[m] in the set T_2 and the common attribute C is defined by repeating this entire process until all V[m] in T_2 have been traversed. By identifying the edges that constitute the shortest path from each

vertex to V[k], the transformation from any attribute to its common representation is composed as a chain of the user-defined SONT-SONT attribute relationships specified in the given sub-graph. There are two key assumptions made in generating these transformations: (i) for every SONT-SONT attribute relationship, a relationship between the data types of those attributes is also defined, and (ii) transformations in relationships between disparate concepts are explicitly defined by the federation developer. Assumption (i) is made when the transformation between the data types of two attributes in an edge E_J is appended to the equation relationship between the input and output variables. As long as that data type level transformation procedure exits, a call can be made to it during the execution of a transformation between V[m] and C. Assumption (ii) may be alternatively stated as—if an explicit transformation between two vertices in an edge E_J is not specified, then the vertices are assumed to be two representations of the same concept. Therefore, an appropriate transformation between their data types adequately captures the traversal from the start to end vertex of all such E_J.

The complexity of the algorithm listed above is of the order $O(N^2)$, where N is the number of vertices in a given sub-graph T_2. The procedure repeats its outer loop N times, so that a transformation is generated for each SONT-Common relationship in the set T_2. The inner loop iterates until all edges in the shortest path from V[m] to V[k] are traversed. If there are N vertices related to each other, the longest path between any two vertices will have $N-1$ edges, such that every vertex in T_2 is visited between V[m] and V[k]. As has been mentioned earlier, the number of related attributes in a sub-graph relates directly to the number of SONTs in the federation. It is improbable that a

prodigiously large number of federates will be part of a federation. Therefore, for all practical purposes, the execution of this procedure will require insignificant time and computing power.

An example to illustrate the functioning of transformation stub generation procedure is provided in Figure 4.6. In the sub-graph defined , vertex V2 is selected to be the common representation. A SONT-Common transformation is to be defined from each vertex to the common representation. The properties of each vertex are listed in the figure. Edge E1 is the only edge that corresponds to a transformation (from attr2 to attr3) that has already been explicitly provided by the federation developer. In the following paragraph, we trace the generation of the *function_to* routine in the SONT-Common relationship between *Attr1* and the common attribute, using the procedure detailed above.

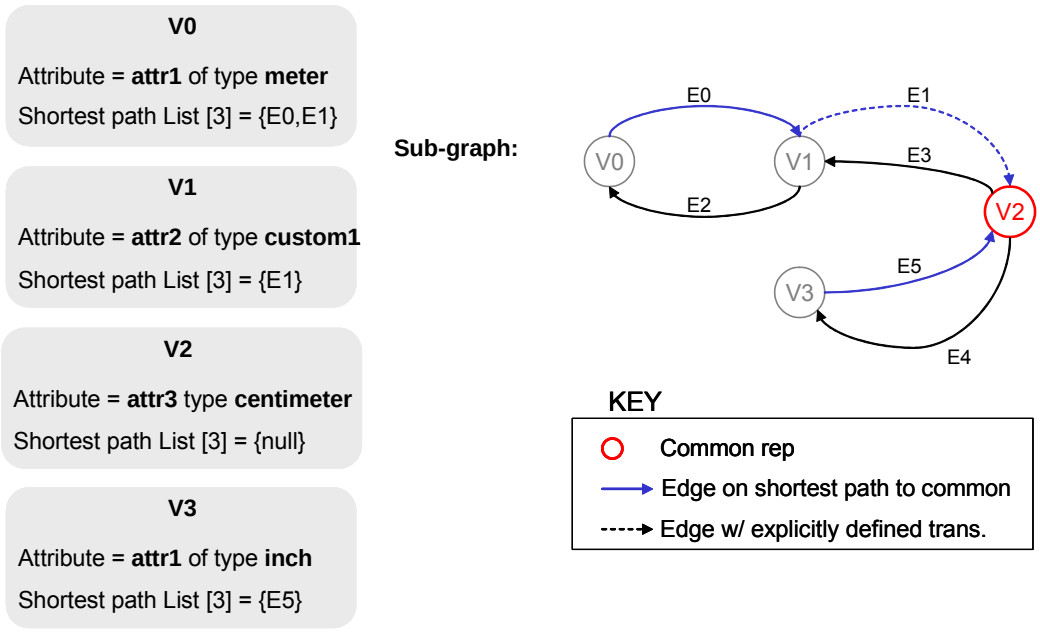


Figure 4.6: Example Attribute Graph to Illustrate Transformation Stub Generation

The procedure begins by selecting V0 to be V[m], following which, a relationship R between V0.attribute (i.e. *Attr1*) and the common representation is identified, and a corresponding transformation header is created for the *function_to* slot of R. This header is as follows:

```
Centimeter attr1_to_Common (Meter input)
   {
     Centimeter output;
     output = input;
```

Following this, the shortest path from V0 to V[k] (V2) is identified as the 3rd index of V0's shortest path list. The first edge in this path is E0, which connects V0 to V1. Given that this edge does not correspond to an explicitly defined transformation, the required transformation from *Attr1* to *Common* is appended as:

```
Centimeter attr1_to_Common (Meter input)
   {
     Centimeter output;
     output = meter_to_custom1(input);
```

The next edge E1 is traversed. This edge corresponds to a user-defined transformation, hence the appended *Attr1* to common transformation is as follows:

```
Centimeter attr1_to_Common (Meter input)
   {
     Centimeter output;
     output = attr2_to_attr3(meter_to_custom1(input));
```

At this point, all edges in the path from V0 to V2 have been traversed. Therefore, the transformation can be closed with a return statement, as follows:

```
Centimeter attr1_to_Common (Meter input)
   {
     Centimeter output;
     output = attr2_to_attr3(meter_to_custom1(input));
     return output;
   }
```

This example highlights a key point with regard to the composition of transformation stubs—given that the data types of both *Attr1* and the Common representation are primitive (they are meter and centimeter, respectively), one may question why a direct transformation cannot be instantiated between these attributes, (based on the fact that the relationship between any two units of a measurable quantity is known from their respective conversion factors). While the data types of both attributes (*Attr1* and *Common*) are primitive, the knowledge as to the concept level relationship between them is not specified. That is, it is not known whether the conversion of *Attr1* to *Common* involves only a conversion between their data types. Furthermore, the shortest path from *Attr1* to *Common* signifies the least lossy transformation from *Attr1* to *Common*; specifying a direct relationship may have implications as to the lossiness of the resulting transformation. Therefore, even though both *Attr1* and *Common* have primitive data types, the transformation between them is composed as a chain of transformations in the shortest available path from *Attr1* to *Common*.

Thus far, only the generation of a transformation stub from a SONT attribute to its common schema equivalent has been discussed. To complete the specification of a SONT-Common relationship, a transformation in the opposite direction, i.e. from the common to SONT representation must be defined as well. The value of *function_from* for a given relationship R between V[m] and C can be determined along side that of *function_to*, following the same set of steps as listed above. To define a transformation in the opposite direction, the shortest path from the vertex V[k] (corresponding to the common representation) to each V[m] needs to be identified. Other than this, the steps undertaken to generate transformations from C to each V[m] remain unchanged.

Having elaborated the specification of transformations between SONT and common attributes of objects and events, a final set of transformations needs to be defined between the various data types of these attributes. Recall that transformations between attributes involved transformations between their data types. For example, a transformation from the *Vehicle* attribute *Position* to its corresponding common representation may involve a conversion from *2-D Coordinate* to *3-D Coordinate*. For the FONT specification to be complete, and for the attribute-level transformations to function correctly, it is vital to define transformations between the set of related data types.

For every relationship between two SONT attributes specified by the federation developer, he or she must also specify the existence of a relationship between their data types. Furthermore, if both these data types are not primitive, matches between the individual attributes of these data types must be specified, as was done with Object and

Event level relationships. Again, if the individual attributes of two related data type classes relate to each other such that a transformation between them only involves a conversion between their own data types, then no further information need be specified, except, of course, the existence of a match between two or more data type attributes. For example, in the relationship between *2-D Coordinate* and *3-D Coordinate*, matches are specified between the individual attributes *X & Abscissa*, and *Y& Ordinate*. However, if the transformation between two data types involves a more complex relationship between their attributes, it must be explicitly specified by the federation developer.

For those non-primitive data type relationships whose transformations are not explicitly defined, a graph based approach can be taken on to generate these transformations in an automated fashion. A directed graph consisting of the set of related data type attributes can be constructed, just as was done for the attributes of objects and events. Using this graph, the transformation between two related data types can be derived via the following procedure:

```
procedure Generate Datatype Transformations {
 for each Relationship R : (R.to, R.from ∈ datatype & R.to, R.from ∉ primitive datatype &
 R ∉ Relationships with explicitly defined transformations) {
   start =R.from
   end =R.to
   Create Transformation Header (R.function_to, start, end)
    for each V[m] : V[m].attribute ∈start {
       for each V[n] :V[n].attribute ∈ end {
          if V[m].shortest distance list [n] ≠ ∞ {
             for (i=0, i <length (V[k].shortest path list [m]), i++){
                 E_j = E [V[m].shortest path list [n] [i]]
                 Find corresponding relationship R_l : R_l is between V[E_j.start].attribute &
                 V[E_j.end].attribute
                 Append Transformation (R_j.function_to.routine,
                 datatype (V[Ej.start].attribute), datatype (V[Ej.end].attribute) )
             }end for
          }end if
       }end for
   }end for
 Close Transformation(R.function_to)
 }end for
}end procedure
```

This procedure identifies the set of relationships R between data types that are not primitive and do not have predefined transformations. For each such relationship, the *from* and *to* data types (named start and end, respectively) are identified, and corresponding header for the *function_to* values of R is instantiated as:

```
<end datatype> <start datatype>_to_<end datatype> (<start
datatype> input)
{
```

Once the transformation stub header has been defined, the conversions between the individual attributes of the start and end data types need to be specified. In the procedure presented above, the set of vertices corresponding to attributes in the domain of the start data type are identified. For each such vertex V[m], if a reachable vertex V[n] is found such that the attribute corresponding to V[n] is in the domain of the end data type, a transformation from V[m] to V[n] is composed by traversing through the n^{th} index of V[m]'s shortest path list, just as was done with the attributes of objects and events. The transformation from V[m] to V[n] is appended to the data type level transformation as:

```
output.<V[n].attribute> = (<EJ's start vertex data
type>_to_<EJ's end vertex data type>
(input.<V[m].attribute>;
```

This process is repeated for each attribute V[m] in the start data type's domain. The resultant transformation stub is of the following form:

```
<end datatype> <start datatype>_to_<end datatype> (<start
datatype> input)
{
   <end datatype> output;
   output.<V[n1].attribute> = (<EJ's start vertex data
   type>_to_<EJ's end vertex data type>
   (input.<V[m1].attribute>;
   output.<V[n2].attribute> = (<Ek's start vertex data
   type>_to_<Ek's end vertex data type>
   (input.<V[m2].attribute>;
   return output;
}
```

In this manner, the transformation between two data types is derived based on the matches specified between their individual attributes. To recap, these transformations are only generated for those data types whose individual attributes relate in a manner such that they are conceptually equivalent but have different representations. All other transformations, except for those between primitive data types, must be specified explicitly be the federation developer. The knowledge required to generate transformation routines between primitive data types is hard-coded as its own set of procedures in GRIT algorithm. As an example, the generation of transformations between unit data types was discussed in Chapter 3 (Section 3.7.2). Procedures to generate transformations between primitive data types are trivial and do not exploit the graph-based representation of a FONT. Hence, there is little value in discussing these procedures in this chapter.

Having defined a set of transformations between the data types of related attributes, an iteration of the overall GRIT algorithm comes to an end. At this point a complete specification of the FONT is available to the federation developer. In the context of the *Vehicle—Car* example, common representations for the attributes *Position* and *Location*, and the objects *Vehicle* and *Car* are generated following the procedure presented in Section. Furthermore, transformation stubs for the SONT-Common attribute relationships are generated as was illustrated in this section. Finally, transformations for the relationship between *2-D* and *3-D Coordinate* data types are inferred using the routine presented above. The common representations for related SONT attributes and the set of SONT-Common composed relationships that involve information loss are reported to the user. If the user is cognizant of a non-lossy transformation to replace an inferred lossy one, he or she may specify this knowledge and initiate another iteration through the GRIT algorithm. The final specification of the FONT is obtained when the federation developer is satisfied with the extent of information loss in end-to-end attribute transformations.

4.6 Assessing the Structural Validity of the GRIT Algorithm

In the previous sections, the detailed specification of the GRIT algorithm has been explored. Having done so, the readers' attention is now focused on the validity of this algorithm. Given that this algorithm is a key component of the overall framework, and is the main avenue by which automation in federation development is supported, its validity must be assessed in order to pontificate about the validity of the body of research presented in this thesis. While we may not be able to say much in the way of the

performance validity of this algorithm at this point, the structural validity and internal consistency of this algorithm is accepted based on the arguments below.

In basing the GRIT algorithm in graph theory, a well developed branch of discrete mathematics, a strong foundation has been laid for the development of a system to support automation in federation development. Graph theory has been formally researched since the early 1930's and since then has been applied to solve problems in several different domains. While the procedures listed in this chapter do not imply the use of a certain programming language, they are firmly grounded in a graph-based representation that has evolved over the years. Also, the application of Dijkstra's algorithm, which is well accepted as an efficient, valid algorithm to traverse paths between vertices in a graph, further bolsters the validity of the GRIT algorithm.

The GRIT algorithm process model depicted in Figure 4.1 helps to validate the fact that this algorithm is internally consistent. Each step in this model builds upon previous one, and the automated specification of the FONT progresses in a serial fashion. The first procedure generates a graph corresponding to a set of related attributes in the FONT. Using this graph, the shortest distance and path between two attributes is noted, which in turn is applied to determine a common representation for a set of related attributes. Finally, transformations are defined between the SONT and common representations using the previously defined shortest path between them. Essentially, each subsequent procedure uses the output of its predecessor as input to perform its tasks. These procedures do not negate or conflict the work of previously executed procedures; they

build upon them. Similarly, the GRIT algorithm as a whole is internally consistent with the other components of the ontology-based federation development framework. It uses the set of relationships previously defined in the FONT to generate a new set of entities and relationships between them; thus building upon user-specified knowledge, not negating or modifying it in any way.

CHAPTER 5

THE DEVELOPMENT OF A FEDERATED AIR TRAFFIC SIMULATION

To validate the usefulness of the framework and GRIT algorithm in supporting federation development, it is important to apply them to an example problem that is representative of the scope of problems they have been designed to address. If we can demonstrate that the constructs we have developed in previous chapters are applicable to solving such an example, an important step towards accepting the validity of the hypotheses proposed in Chapter 1 is accomplished. In this chapter, the ontology-based federation development framework is applied to support the automated generation of an air-traffic federated simulation. In the context of the design of an airport, we introduce the air traffic federated simulation scenario and explicitly state why this example problem is apt to study the application of the framework and algorithm. The framework process model defined in Chapter 3 is then followed to develop the federated simulation. Based on the resultant FONT, the effectiveness of the framework in supporting automation in achieving representational compatibility in a framework is established and discussed in detail.

5.1 Introduction to the Federated Simulation

We employ a federated air traffic simulation to test the performance validity of the framework and algorithm developed in previous chapters. This simulation is developed in order to study the behavior of several sub-systems that are part of, or interact with an airport being designed. The characteristics of this federation development problem and the extent of interplay between federates makes this an ideal case study to support the validation of our hypotheses. In the following paragraphs, the simulation scenario, the goal of the federated simulation and the individual federate models are introduced.

The simulation of aircraft traffic at and around an airport is conducted in the context of an airport design problem. Consider that a new airport is to be developed for a metropolitan city. This airport is a very large system consisting of multiple components such as terminals, runways, control towers and hangars. Aside from physical components, there are several functional sub-systems that are part of an airport, including air traffic control, aircraft servicing and several passenger related services. To design such a large system, the 'Vee' model for system engineering is employed (Forsberg and Mooz 1992). In accordance with this model, the design of the entire airport system is broken down into several smaller design problems. Specifications are developed for the coupled individual systems, which are then designed by separate teams. As these systems are being designed in parallel, it is necessary to ensure that their design progresses in a manner such that the expected behavior of the entire system adequately addresses its requirements. Therefore, it becomes necessary to simulate and analyze the behavior of the airport system as a

whole. To do so, the simulation models corresponding to individual sub-systems of the airport need to be integrated and executed in a federated fashion. We employ the use of the ontology-based framework to support the automated integration of these simulation models into a federated simulation.

The federated simulation being considered in this chapter is employed at the early stages of the airport system development process. Upfront, some basic characteristics of several sub-systems need to be defined before detailed design can be investigated. For example, to define a specification for the network of runways at an airport, the required number of runways needs to be identified first. These requirements stem from the overall requirements of the airport, such as the volume of air traffic that the airport system is expected to manage. In this example, the overall goal of the federated simulation is to help designers identify a set of design-to specifications for the individual sub-systems of an airport. This federated simulation provides information about the expected behavior of the airport system, based on the characteristics of its individual sub-systems. By studying the behavior of the airport with regard to managing different air traffic scenarios, the specification of individual sub-systems can be decided upon.

Given the overall application scenario and goal of the air traffic federated simulation, let us further investigate the airport system in terms of its constituent sub-systems. The airport system, as a union of its individual components, is illustrated in Figure 5.1. An important sub-system of an airport is the air traffic controller (ATC), which is responsible for queuing aircraft in departure and arrival corridors and for directing aircraft in the

airport's airspace to land. It is vital to make sure that the traffic control procedures in this system are designed to handle a large number of aircraft in the airport's local airspace. A similar component is the ground traffic controller (GTC), which is responsible for managing aircraft traffic between gates, hangars and runways. Related to these sub-systems are the runway networks and taxiway networks on which aircraft travel. It is important to make sure that myriad aircraft can travel on these networks, and that they are designed to allow smooth flow of a large number of aircraft (as has been projected). Finally, the gates at which aircraft are parked, the resources they require to deplane and board passengers and cargo and the inter-flight services performed on aircraft constitute the ground-based aircraft services component of the airport. While there are several other systems that are part of an airport, in this example, we limit ourselves to the set of systems defined above.

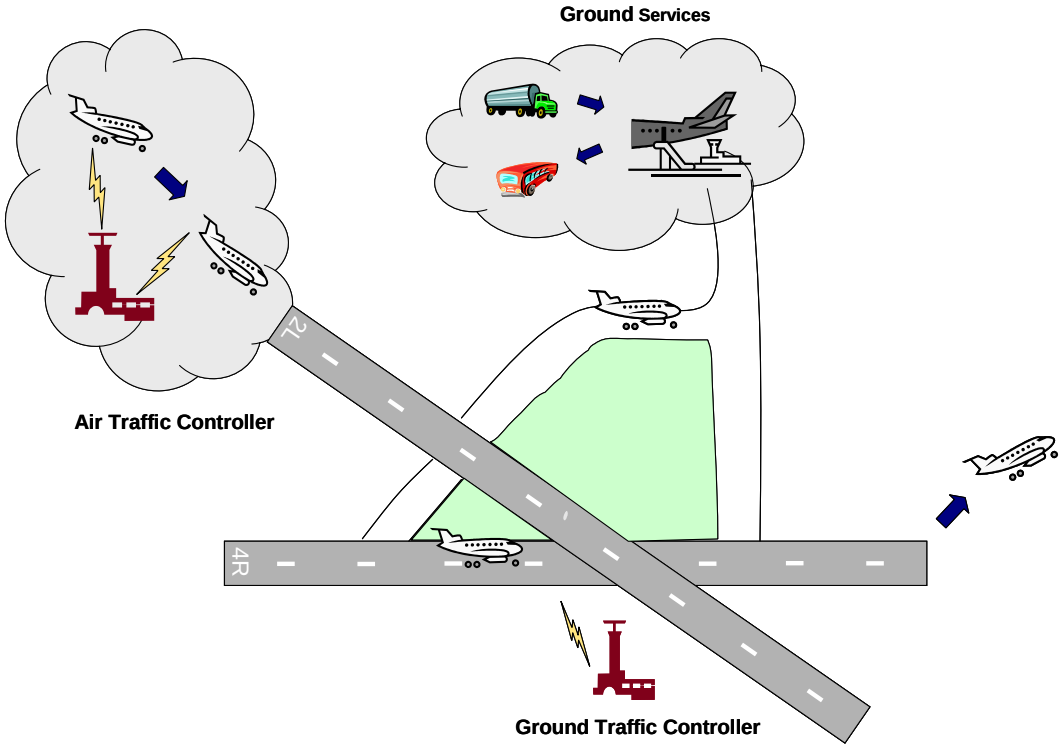


Figure 5.1: Sub Systems in an Airport System Design

The simulation of the aircraft system as a whole is conducted to identify the key design-to specifications of its sub-systems. By subjecting the airport system to different volumes of aircraft landing, taking-off, refueling and so on, the effectiveness of the different sub-systems in managing these aircraft can be studied. Based on this, the specification of each sub-system can be modified until the airport system as a whole behaves such that it adequately performs aircraft management tasks defined as part of its requirements. The goals of conducting this federated simulation experiment, with respect to each sub-system, are as follows:

- To gain insight as to the efficiency of the air traffic controller in directing and queuing aircraft for landing: By recording information such as the average time an aircraft spends in the airport airspace, waiting to land, and the fuel consumed in doing so, the effectiveness of the ATC, given its current specification, can be gauged. Note that other sub-systems also affect the ATC's performance, such as number of runways and the usage of those runways by the GTC. Therefore, it is important to simulate the behavior of these sub-systems as a whole.

- To gain insight as to the effectiveness of the ground traffic controller in managing the flow of aircraft between gates and runways: Just as with the ATC, the effectiveness of a given GTC specification can be gauged based on the results of the federated air traffic simulation. The average time an aircraft spends waiting to take-off or park at a gate are indicative of the GTC's performance, but in turn depend on other sub-system parameters, such as the number of gates at the airport.

- To determine parameters of ground-specific aircraft services: The air traffic simulation at the airport helps designers to decide on an adequate number of gates, ground crew, re-fueling tankers and so on.

Designers arrive at the best suited or satisficing specification of the individual sub-systems of the airport in an iterative fashion. By running the system-level federated simulation and analyzing the emergent behavior of the system, required changes in the

specification of each sub-system can be determined. Once these changes are implemented, the simulation can be executed again, repeating the process until the system level behavior is acceptable.

Corresponding to the sub-systems identified above, there are three federate simulation models that comprise the air traffic federation. These are (i) the Air Traffic Controller (ATC) model (ii) the Ground Traffic Controller (GTC) model and (iii) the Ground Services model. The ATC simulation model is a representation of the system that manages aircraft traffic in the airport's airspace. This system keeps track of every aircraft in the vicinity of the airport and sends and receives messages from different aircraft. The GTC simulation model represents the system in place to control the flow of aircraft traffic on ground. This system keeps track of the position and status of all aircraft on the ground, and communicates to aircraft via messages, instructing pilots to park, taxi to a certain location, take-off and so on. Finally, the ground services federate models the use of on-ground resources by different aircraft that arrive at the airport, such as the availability of gates and use of ground personnel to deplane passengers and offload cargo.

Given that the airport design is still in its early stages, our interest is in studying the behavior of the airport at a relatively high level of abstraction. Specifically, we want to gauge how this system behaves in response to distinct events where different volume of aircraft arrive and depart. Therefore, each federate in the air traffic simulation federation corresponds to a discrete event simulation model. These simulations model their respective systems such that they are defined to be in a particular state at a given time

stamp, which may change after a discrete interval (as opposed to in a time-continuous fashion). Based on this, interactions between the federate simulations takes place in an event-driven fashion, wherein the change in state of an entity in one federate simulation may trigger a change in another. The specifics of the interaction between the four federates of the air traffic simulation is explained as follows.

Given the interactions between the individual sub-systems of the airport, there is significant interaction between the individual federate simulations in the air traffic federation. In the real system, the ATC and GTC systems will intermittently contact each other and the aircraft (pilots). Similarly, when a given aircraft instance in the ATC simulation lands on a runway, the GTC simulation must be made cognizant of this event. Every aircraft that lands is sent a message by the GTC, indicating a destination gate and a taxiway to follow. Therefore, the GTC must keep track of all available gates, and all taxiways upon which aircraft are traveling at a given time. The Ground Services simulation model captures all gate related resources. This federate communicates the availability of gates to the GTC. Furthermore, the appropriate number of ground personnel and re-fueling tanks are assigned to work at a given gate based on the payload and fuel content of a given aircraft parked at that gate. Therefore, there is interplay between the GTC and Ground Service federates that involves exchange of payload and fuel data. The complete interoperation between the federate simulations (at a high level of abstraction) is listed in Table 5.1.

Table 5.1: Interactions between Federate Simulations in the Air Traffic Federation

Involved Federate Simulations	Description of Interaction
ATC, GTC	When an aircraft in the ATC simulation lands, information about its call sign, the type of aircraft, its fuel level and so on need to be published to a corresponding aircraft in the GTC simulation. Similarly, when an aircraft in the GTC simulation takes-off, this information needs to be subscribed to by a corresponding aircraft instance in the ATC federate.
ATC, GTC, Ground Services	Each federate models a runway that are part of the airport. The availability of a runway to land on (in the ATC), to take-off from (in the GTC) or on which maintenance needs to be conducted (in ground services) and needs to be shared across all three federates. If the GTC assigns an aircraft to take-off from a runway, the state of that runway changes to in-use. Other federates must subscribe to this change. Similarly, if the ATC or ground services updates the state of a runway, all other federates must subscribe to this state-change.
GTC, Ground Services	The GTC maintains information about gates, as to whether they are in use or free. Corresponding to this, there are gates modeled in the ground service federate. When the GTC assigns an aircraft to park at a given gate, this needs to be reflected in the ground services federate as well.
	The fuel level and payload of an aircraft at a given gate in the GTC simulation must be communicated to the ground services federate. When the aircraft is re-fueled in the ground services simulation, the fuel level of a corresponding aircraft in the GTC simulation must be updated.
	When an aircraft in the ground services federate has been prepared for take-off, an event must be triggered in the GTC to queue the aircraft for departure. When the aircraft clears the gate in the GTC federate, the state of a corresponding gate in the ground services federate must be updated.

Having described the federated air-traffic simulation in terms of its purpose and constituent federates; we may proceed to applying the ontology-based framework to support the development of this federation. However, before doing so, the validity of

using this example problem to gauge the performance of the framework must be accepted; else this exercise is futile. In the following section, an argument as to why this problem is apt to demonstrate the application of the ontology-based framework for federation development is presented. Once we have established the empirical structural validity of this example problem, we proceed with the development of the associated simulation ontologies.

5.2 Empirical Structural Validation

The goal of applying the ontology-based framework to support the integration of the airport-related federate simulations is to study the applicability of the framework in the context of an example that is representative of the type of federation development problems it is meant to address. Given this goal, it is necessary to determine if the air traffic simulation problem is an apt test case. In order to do so, we pose the following questions about this example—(i) Is the federated air traffic simulation representative of the types of problems this framework has been designed to address? (ii) Does this example allow us to validate specific characteristics of the framework? By answering these questions in the following paragraphs, we determine that the air traffic federated simulation *is* an appropriate test case.

In Chapter 1, the application scope of this framework has been elaborated so as to pertain to system-level simulations of large scale engineering systems. Traditionally, sub-system level simulation models exist, which need to be integrated with each other in order to study the emergent behavior of a system. It is within this context that the ontology-based

framework is to be applied to support achieving interoperability between sub-system level simulations in an automated fashion. Clearly, the air traffic simulation is a system-level experiment to study the expected behavior of the airport as a whole. As has been detailed above, there are several sub-systems to an airport, each with corresponding behavior models, that need to interoperate with each other. Since these sub-system models are all discrete event models, the interaction between them must be carried out in a message-passing form at discrete time steps. This is a typical federated simulation problem, which is exactly the type of simulation interoperability problem our framework is designed to address. Therefore, at a high-level, it is evident that the air traffic example is representative of the federation development problems within the scope of this framework.

At a lower level of granularity, we must investigate if this example problem allows us to determine certain key characteristics of this framework. These characteristics and the extent to which they can be studied using this example are elaborated as follows:

- Expressiveness of the World Ontology: In hypothesis 2, the definition of a metamodel for capturing simulation concepts in an ontology has been proposed, which has then been realized in the development of the World Ontology. To validate this hypothesis, it is important to show that this metamodel can be used to express myriad shared simulation concepts in a SONT. The air traffic example federated simulation is a good example to test the expressiveness of the World Ontology. The individual federates in the air traffic simulation involve a diverse

set of concepts, including physical objects that are persistent through the simulation, such as an aircraft, runways and gates. There are also several non-physical, non-persistent concepts modeled in each federate, such as the communication between the ATC and GTC. Finally, these simulation models include several parameters and state variables including those that capture the status of gates and runways (e.g.: free, busy, open, closed) that need to be shared across federate domains.

- Expressiveness of relationships: Along with the ability of the World Ontology to represent shared concepts in a given simulation domain, the ability to express different relationships between these concepts in an ontology must also be tested. Again, the air traffic control is a quintessential with which this can be tested. Given that there is significant interaction between individual federates, several relationships will need to be defined between entities of each federate. These relationships range from simple equivalence to complex mappings. For example, both the GTC and ATC simulation models include the concept of an aircraft's heading, measured as an angle, in degrees and minutes. The relationship between these two concepts is one of pure equivalence. A more complex relationship exists between the fuel level of an aircraft parked at a gate, its fuel capacity and the number of re-fueling tankers required at a gate.

- Correctness of the GRIT algorithm: A graph based approach to automate the generation of transformation stubs between related entities in a federated

simulation is proposed in hypothesis 1, and implemented in the GRIT algorithm. In order to validate this hypothesis, we must investigate whether the GRIT algorithm is able to correctly derive transformations between related entities. In the air traffic federated simulation example, several equivalent concepts with disparate federate representations are related. Subsequently, mappings between a subset of federate entities involve information loss. For example, the identifier for a runway in the ATC simulation includes the runway number as well as its associated Instrument Landing System (ILS) code. This identifier relates to runway ID's in the GTC which is defined only in terms of the runway number. This and other relationships involving lossy transformations can be used to investigate whether the GRIT algorithm instantiates the appropriate common schema. Subsequently, the transformation stubs composed between federate and common schema entities in the air traffic federation can be evaluated to determine if the GRIT algorithm correctly derives these procedures.

Based on the argument above, the empirical structural validity of the hypotheses proposed in Chapter 1 is accepted. In other words, we accept that the selected example problem i.e. the air traffic federated simulation is appropriate to demonstrate the intended use of the ontology-based framework. Having done so, we may proceed with the application of the framework to this federation development problem, based on which the performance of this framework can be analyzed.

To employ the ontology-based framework and GRIT algorithm for the development of the air traffic federated simulation, we follow the framework process model prescribed in Figure 3.2. A flow chart indicating these steps in the context of this example federation development problem is illustrated in Figure 5.2.The first step in this process is to create simulation ontologies corresponding to each federate in the air traffic federation. These SONTs should describe the federate representations of shared objects, event, attributes and their data types. Once the SONT corresponding to the ATC, GTC and Ground Services simulations are available, an initial FONT is to be generated that includes all SONT domains. At this point, we (the federation developers) specify relationships between the coupled entities of the three SONT domains, based on the interactions enumerated in Table 5.1. Note that for every object or event level relationship that is specified, relationships between their individual attributes, and the data types of those attributes must be instantiated as well. For each attribute-level relationship, we must specify knowledge as to the lossiness of the transformations between the related attributes. Having specified the complete set of SONT-SONT matches, the GRIT algorithm is invoked to generate the required common information model and SONT-Common transformation stubs. This algorithm creates a graph for related attribute, objects and events, and uses the shortest distance or equivalently the least lossy chain of relationships between related attributes to determine their common representation. The required transformations are then generated using knowledge of the shortest path between the related attributes. Finally, when the GRIT algorithm has finished it execution, we may examine the resulting common information model and the resultant lossy transformations, and revise them if required.

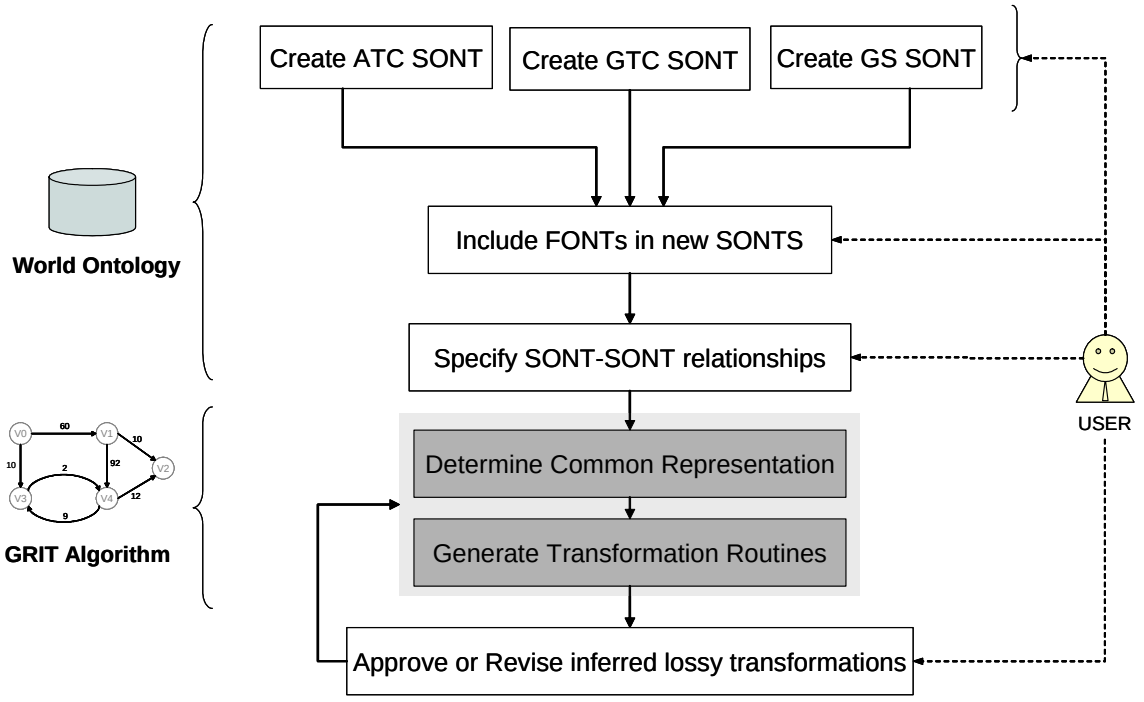


Figure 5.2: Ontology-based Framework Application Process for Air Traffic Federated Simulation

5.3 Development of SONTs

In accordance with the figure above, we begin the air traffic federation development process by developing SONTs for each of the three federate simulation models. The shared concepts in these federate domains are modeled in terms of the common vocabulary defined in the World Ontology, as follows:

5.3.1 The Local Air Traffic Control (ATC) SONT

The air traffic control federate simulates the management of aircraft in the airport's local airspace. In this simulation, the local airspace is populated with different volumes of

aircraft in a transient manner, which are queued to land at different runways of the airport. The queuing system assigns instructions to each aircraft so as to allow them to land in a timely and safe manner (without having to maintain a holding pattern around the airport for too long or collide with other traffic). Furthermore, the management of outbound aircraft traffic is also modeled in this federate.

There is a long list of individual concepts represented in this simulation model, but our interest is only the simulation's interface—those concepts that are shared with other federates in the air traffic simulation. A central concept in the ATC simulation is that of an aircraft. During the simulation execution, several aircraft simulation entities are created and destroyed as they enter and leave local airspace. Each aircraft is described in terms of its type (turbo prop, business jet, twin engine commercial jet, 3+ engine commercial jet), its position , which include latitude and longitude (as angles) and altitude (in feet above mean sea level), heading (in degrees), payload (in metric tons), and fuel content (in Imperial gallons). There are other attributes of aircraft captured within this simulation (such as speed of aircraft and average rate of descent), but again, they are of little consequence given that they are not shared with other federates. The creation and deletion of aircraft in the simulation is triggered by two events, namely *New Aircraft in Airspace* and *Aircraft on Ground*, respectively. The second key shared concept in the ATC simulation is that of a runway. Each runway is defined in terms of its length, ILS (instrument landing system), Localizer frequency (in MHz) (which is used to line the aircraft laterally with the runway centerline), and its availability at a given time (in-use or

free). This information about runways is used to determine which planes can land on which available runways at a given point in the simulation.

Having introduced the key concepts of the ATC simulation that are shared in the air traffic federation, let us now investigate how they are modeled in a simulation ontology. Every shared concept that is persistent throughout the length of the simulation should be modeled as an Object (instance of the object metaclass) and every non-persistent concept as an event (instance of the event metaclass). Given that aircraft are created and destroyed, they are not fully persistent. However, within the simulation, they are maintained for more than a single time stamp. Internally, there is some notion of persistence associated with aircraft, but the only time aircraft information is exchanged with the GTC is when an aircraft is created or destroyed. Therefore, we choose to describe the aircraft concept in terms of two events: *New Aircraft in Airspace* and *Aircraft on Ground*. Each event is described in terms of its constituent attributes (instances of the attribute metaslot). The attributes of the *Aircraft on Ground* correspond to those attributes that are to be published to the GTC simulation when an aircraft lands. The *New Aircraft in Airspace* event consists of attributes that are subscribed to from the GTC when an aircraft takes-off, plus other attributes required to initialize a new aircraft. That being said, it is likely that some attributes may be shared between the two events. Therefore, a higher-level, abstract aircraft event can be defined, such that its attributes are subsumed by both the *Aircraft on Ground* and *New Aircraft in Airspace events*. The resultant hierarchy of events (and the attributes in their domain) is illustrated in Figure 5.3. Several attributes of these events (heading, payload, fuel content) have value-types

that are units of measurement, which have been previously defined in the World Ontology. However, two attributes, *Position* and *Fuel Content* require custom data types to be defined. In the ATC simulation model, the fuel content of an aircraft is defined via a complex data type that includes both the total fuel carrying capacity of the aircraft and the amount of fuel currently on-board. In a similar fashion, a new data type *Fuel Content Type* is instantiated within the SONT, with two attributes, namely *Capacity* and *Remaining Fuel*, both having unit data type *Imperial Gallon*. Further, the value type of the *Fuel Content* attribute in the domain of the aircraft events is set to *Fuel Content Type*. Similarly, a new data type (*Lat-Long-alt*) is instantiated to capture the value of the *Position* attribute. This data type has three attributes, *Latitude* and *Longitude*, both of the degree-minute data type for angle measurement, and *Altitude* of unit data type *Foot*.

Runways are persistent through the length of the simulation and are modeled as objects. In the ATC simulation, each runway is characterized by an alpha-numeric number that specifies the runway and the direction it is to be approached from (e.g.: 4R, read runway four-right), an ILS Localizer frequency, the runway's length and its current usage status. In the ATC SONT, individual attributes are instantiated corresponding to each of these fields. All these attributes have primitive data types, which makes the specification of the runway object relatively trivial. Specifically, *Runway Number* takes on the *String* data type, *Length* is specified in feet, *ILS Localizer* is specified in MHz, and the status of the runway is captured as the attribute *Is_Available* of type *Boolean*. The resultant ATC SONT is illustrated below.

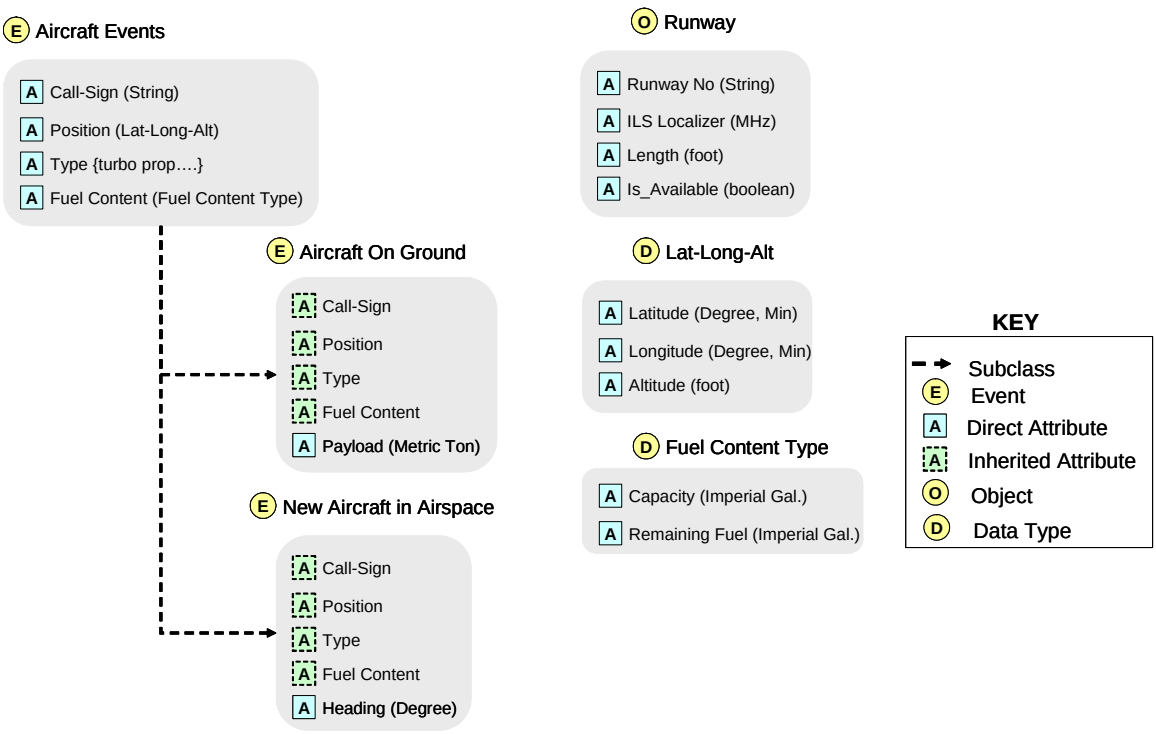


Figure 5.3: Air Traffic Control SONT Specification

5.3.2 Ground Traffic Control (GTC) SONT

The ground traffic federate simulates the direction of all aircraft traffic on ground. In this simulation, new aircraft are created as they land on runways. They are each assigned to follow specific taxiways to assigned gates. There are multiple aircraft in existence in the simulation at a given time stamp, each of which can be in a different state (e.g.: parked, landed, and taxiing). Therefore, the focus of this simulation is to model the behavior of the ground controller in making sure traffic flows smoothly, and without accidents.

Since the ground controller interacts with aircraft, runway networks, taxiway networks and terminal gates, these are all central concepts modeled in the GTC federate. Of course, in the context of this federation, only runway, aircraft and gate information is shared with other federates. Each aircraft has a number of properties associated with it, such as its location (latitude and longitude), heading (all in degrees, minutes), fuel level, fuel capacity (both in US gallons) and payload (in metric tons). Furthermore, individual aircraft are identified by their call-sign and type (propeller, small jet, small commercial jet, large commercial jet). In the GTC, there are four aircraft related events that are published or subscribe to other federates—Aircraft Landed, Aircraft Parked, Aircraft Chocks Off (indicates chocks removed from tires and aircraft is ready to depart) and Aircraft Departed. Analogous to the ATC, runways are modeled in the GTC simulation with properties Dimensions (Length and Width in feet), and availability (in terms of a true /false Boolean). Finally, the GTC maintains information about gates, as to their location (a sector number) and availability (also a Boolean). It should be noted that while taxiways are an important concept in the GTC, we do not explore them in detail as they are not shared with other federates in the air-traffic simulation.

As with the ATC SONT, aircraft in the GTC SONT are modeled as events. Not that it is wrong for them to be modeled as objects (technically, aircraft are persistent for more than a single time stamp in both simulations), but we choose to model them as events because aircraft information is only exchanged when an event occurs. As mentioned earlier, there are two pairs of Aircraft Events, each of which correspond to Events in the ATC, and Ground Service SONTs, respectively. Similar to the ATC, a baseline *Aircraft Event* class

is created, whose attributes are those that are common to the individual events. These common attributes include *Fuel Level*, *Fuel Capacity* (with data type *US gallon*), *Call-sign* (with data type *String*) and *Aircraft Type* (with an *Enumerated* data type). Other attributes specific to each event include *Heading* (in degrees) and *Payload* (of unit data type *Metric Ton*).

Both runways and gates, which are persistent throughout the simulation, are represented in the GTC SONT as Objects. The *Runway* object is defined in terms of the attributes *Dimensions* (in terms of a custom data type with attributes *Length* and *Width* in feet), *Runway Number* (a String) and *In_Use* (which a Boolean). In a similar fashion, the Gate object has attributes *Gate Number*, *Sector* (both Strings), and *In_Use* (this is a single attribute with two domains, runway and gate). The resultant GTC SONT is depicted below in Figure 5.4.

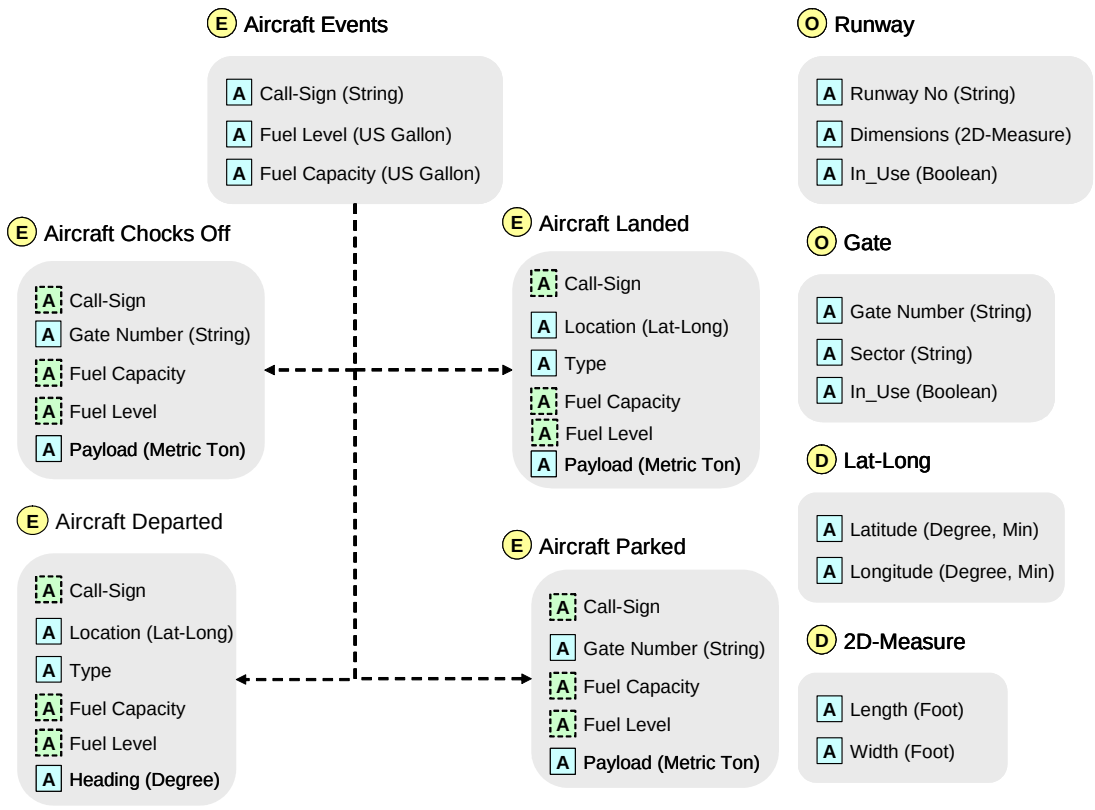


Figure 5.4: Ground Traffic Control SONT Specification

5.3.3 Ground Services SONT

The final federate simulation in the air traffic federation models the behavior of myriad ground-based sub-systems that perform activities centered on aircraft arrival and departure. These activities include deplaning passengers, offloading cargo, re-fueling aircraft and de-icing or maintaining runways as required. Essentially, this federate manages crew and resources as aircraft arrive and depart from gates. When an aircraft arrives at a given gate, an appropriate number of ground-crew members (based on the payload of the aircraft) that are not currently engaged in other tasks are sent to offload

cargo and passengers. Similarly, fuel tankers, cargo trolleys and so on are assigned to attend to different aircraft existing in the simulation.

Gates, runways, ground crew and refueling tanks are all key concepts in the ground services simulation model domain. Of these concepts, those that are shared with others in the federation are runways and gates. The ground services simulation does not explicitly model aircraft as persistent objects; the arrival of an aircraft at a gate is treated as a discrete event, namely *Service Required*, based on whose parameters, the appropriate number of persistent resources (fuel tankers, crew and so on) are set to a 'busy' state. As one might imagine, the state of these resources is changed back to 'free' when another event occurs (*Service Completed*), signaling the aircraft's departure. The *Service Required* event is defined in terms of a set of parameters that include the total payload that needs to be offloaded (in pounds), and the amount of fuel to be supplied (in US gallons), and the gate number at which these resources are to be supplied. The Service *Completed* event is defined solely in terms of a gate number. Gate information, which is maintained throughout the length of the simulation, includes the gate number, location (a sector number) and the status of the gate (busy or free). Finally, runways are modeled in a fashion similar to that of other federates. Each runway is described in terms of its number, ILS localizer frequency (in MHz) (runway service teams are responsible for setting and maintaining on-ground ILS beacons), its dimensions (in feet) and status (busy or free).

Based on the representation in the ground services simulation, all aircraft related information in the ground services SONT are modeled in terms of two event classes—*Service Required* and *Service Completed*. The *Service Required* event has attributes *Payload* (of data type *Pound*), *Fuel Required* (in US gallons) and *Gate Number* (of data type *String*), while Service Completed only reports a *Gate Number* at which an aircraft has been serviced. Runway and Gate concepts are modeled as objects, given that they are persistent through time in this simulation. Each Runway object has attributes *Runway Number* (a String), *ILS Localizer Frequency* (in MHz), *Dimensions* (of a complex data type with *Length* and *Width* as attributes) and *Status* (with an Enumerated data type {busy, free}). The Gate object is defined in terms of attributes *Gate Number*, *Sector* (a String) and *Status*. The complete ground services SONT is illustrated in Figure 5.5.

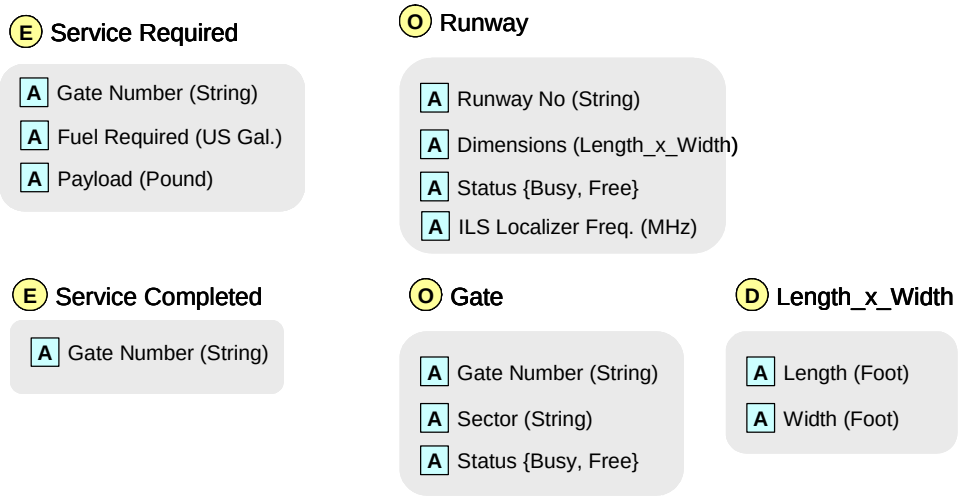


Figure 5.5: Ground Services SONT Specification

At this point, the shared entities in the ATC, GTC and Ground Services federate simulations have been captured in their respective SONTs. It is interesting to note, that in each of these simulation models, concepts are described in an object-oriented fashion, which makes the development of their corresponding SONTs quite straightforward. In general, legacy simulations that do not employ an object oriented representation may need to be federated. For such a federate, an object-oriented interface to the underlying simulation must be defined, based on which the corresponding SONT is modeled.

Having completed the SONT development process for the air traffic federation, we proceed to the definition of relationships between the entities defined in each SONT, in the following section.

5.4 Specification of Relationships

In accordance with the federation development process model illustrated in Figure 5.2, the next step in the development of the air traffic federation is to specify relationships between entities in each SONT domain. To capture relationships (instances of the relationship class defined in the World Ontology) between SONT entities, they must all be included in a single federation level ontology, namely the FONT. Therefore, a new ontology is created such that its domain spans those of the ATC, GTC and Ground Services SONTs. Within this ontology, a set of relationships between different SONT objects and events, their attributes and the data types of those attributes are to be specified. This task is divided into two sections, one to define relationships between ATC and GTC entities, and the other to relate GTC and Ground Services Entities.

5.4.1 Relationships between ATC and GTC Entities

The ATC and GTC federates mainly exchange information about aircraft that land and take-off from the airport. Specifically, when an aircraft in the ATC lands, a corresponding aircraft must be instantiated in the GTC federate. Similarly, when an aircraft in the GTC takes-off, a corresponding new aircraft must be instantiated in the airspace modeled by the ATC federate. In other words, the occurrence of the *Aircraft on Ground* event in the ATC must be published for subscription by the *Aircraft Landed* event in the GTC, and the *Aircraft Departed* event in the GTC must be translated to the *New Aircraft in Airspace* event in the ATC. Therefore, a relationship (instance of the relationship class) is instantiated to indicate that the two events match each other.

When an event or object level relationship is defined, matches between their attributes need to be specified as well. For each pair of equivalent attributes in the *Aircraft on Ground* and *Aircraft Landed* events, a relationship must be specified between them. The attributes of the ATC event *Aircraft on Ground* subscribed to by the *Aircraft Landed* event are *Heading, Position, Payload, Fuel Content, Call-Sign* and *Aircraft Type*. In the same order, these attributes relate to the GTC federate attributes *Heading, Location, Payload, Fuel Level* and *Fuel Capacity, Call-Sign* and *Type*. In each relationship, the corresponding attributes are specified in the *to* and *from* slots of the relationship instance, and knowledge as to whether either transformation (*from* to *to* or *to* to *from*) is lossy must be provided as the value of the *is_lossy* slot. A relationship specified from *ATC_heading* to *GTC_heading* is depicted in Figure 5.6. There is no loss of information in a transformation between these attributes; hence *is_lossy* is set to false in both the

function_to and *function_from* slots of this relationship. Since both attributes have the same pre-defined data type (degrees), no data type level relationship needs to be specified. The same case is observed in the relationship between *ATC_Call-Sign* and *GTC_Call-Sign*, and the relationship between *ATC_Payload* and *GTC_Payload*.

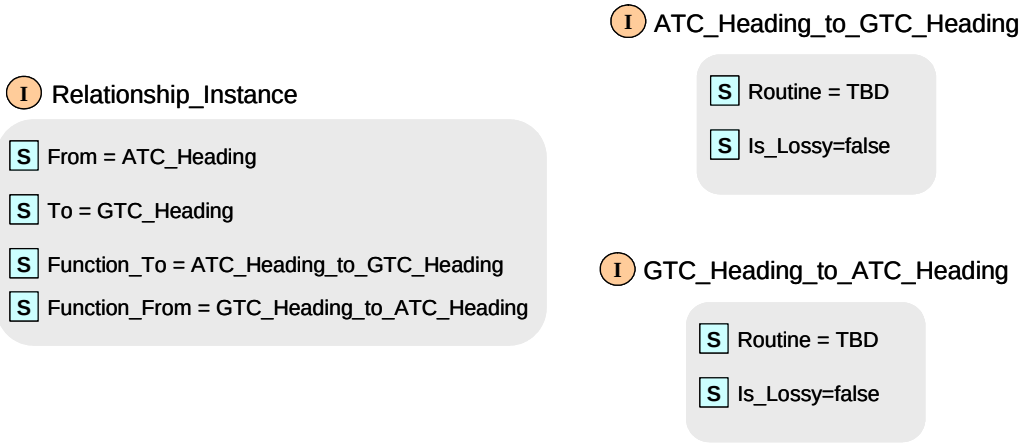


Figure 5.6: Example Relationship Instantiation

Both the ATC attribute *Position* and its related GTC Attribute *Location* have custom data types (*Lat-Long-Alt* and *Lat-Long*, respectively). Since a transformation between them includes a transformation between their data types, a relationship between these data types must be specified as well. The relationship between the *Lat-Long-Alt* and *Lat-Long* data types involves an equivalence mapping between their respective latitude and longitude attributes. Therefore, relationships are defined between these attributes, based on which the GRIT algorithm infers required data type level transformations between *Lat-Long-Alt* and *Lat-Long*. Note that since *Lat-Long-Alt* includes information about altitude while *Lat-Long* does not, the transformation from Position to Location involves

loss of information. This knowledge is captured in the relationship between them as the value of the appropriate *is_lossy* slot.

The relationship between the ATC attribute *Fuel Content* and the GTC attributes *Fuel Level* and *Fuel Capacity* is not quite as straight forward. *Fuel Content* encapsulates both the total fuel capacity of the aircraft and the amount of fuel left at a given time stamp within its own data type *Fuel Content Type*, but the same is not true in the GTC SONT. Therefore, a relationship needs to be specified between a single ATC attribute and two GTC attributes. In accordance with the steps to be undertaken for specifying *n:m* relationships outlined in Section 3.5.2, an aggregate data type *Aggr_Fuel_Cap_Datatype* is instantiated in the GTC SONT, with the GTC attributes *Fuel Capacity* and *Fuel Level*. Subsequently, an aggregate attribute *Aggr_Fuel_Cap_Attr* is defined in the domain of the *Aircraft Events* event in the GTC (Figure 5.7). A relationship is then defined from *Fuel Content* to *Aggr_Fuel_Cap_Attr*. Since these attributes have disparate, custom data types, a relationship (match) must be specified between *Fuel Content Type* and *Aggr_Fuel_Cap_Datatype*. Since the attributes of these data types are conceptually equivalent, the transformations to convert between the two can be generated automatically by simply specifying relationships between the attributes of these data types.

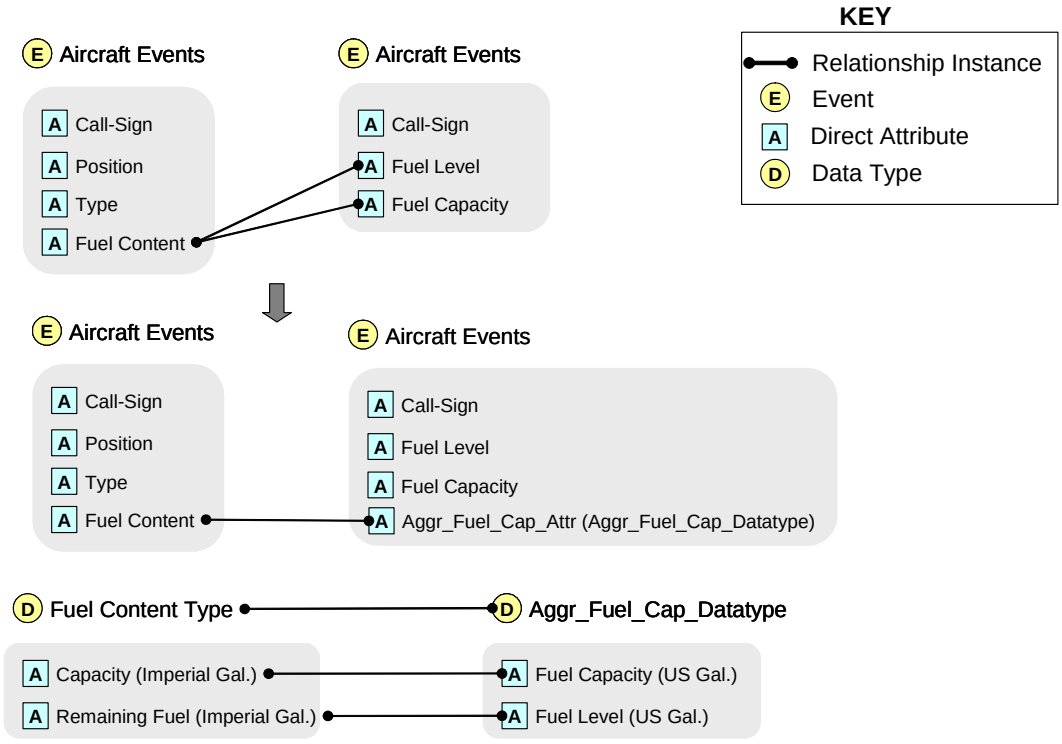


Figure 5.7: Relating the ATC Fuel Content Attribute to GTC Fuel Level and Fuel Capacity Attributes

Another attribute level relationship of interest is that between the ATC attribute *Aircraft Type* and the GTC attribute *Type*. Both these attributes have enumerated data types, and a relationship between them involves specifying a relationship between each pair of enumerals (e.g. turbo prop in ATC is equivalent to propeller in GTC). Given that relationships can only be specified between simulation entities and not enumerals, we cannot capture knowledge of the relationship between enumerals in the FONT. Hence, as part of the relationship between *Aircraft Type* and *Type*, the transformation stubs to convert values between the two must be explicitly defined at this point. The procedure (stub) to convert the value of *Aircraft Type* to a corresponding value of *Type* is captured

in the ontology as the value of *function_to* in the relationship from *Aircraft Type* to *Type*. This procedure is as follows:

```
String Aircraft_Type_to_Type (String input) {
   String output;
   Switch (input){
      Case: “turbo prop” {output = “Propeller”; break ;}
      Case: “business jet” {output = “Small Jet”; break ;}
      Case: “Twin Engine Commercial Jet” {output = “Small
              Commercial Jet”; break ;}
      Case: ”3+ Engine Commercial Jet” {output = “Large
              Commercial Jet”; break ;}
   };
   return output;
}
```

In a similar fashion, the transformation from *Type* to *Aircraft Type* is also specified explicitly.

We summarize the relationship between other events, objects of the ATC and GTC SONTs as follows: The relationship between events *New Aircraft in Airspace* and *Aircraft Departed* involves mappings between their attributes, many of which have already been related, as explained above. An Object level relationship is defined between the ATC and GTC *Runway* objects. Subsequently their related pairs of attributes are *ATC_Runway Number* & *GTC_Runway Number*, *Length* & *Dimensions* and the Boolean attributes *Is_Available* & *In_Use*. It should be noted the *Length* and *Dimension* do not refer to the same concept, hence the transformation stubs associated with their

relationship cannot be automatically generated. These must be specified explicitly along with the definition of this relationship. Based on the directions provided in Section 3.6.2, the required transformations are specified explicitly by the federation developer as follows:

```
Function_to.routine:
2D_Measurement Length_to_Dimension (foot input) {
   2D_Measurement output;
   output.length = input;
   output.width = 0; //User-defined default
   return output;
}

Function_from.routine:
Foot Dimension_to_Length (2D_Measurement input) {
   Foot output;
   output = input.length;
   return output;
}
```

Furthermore, a transformation from *Dimensions* to *Length* involves a loss of information (information as to the width of the runway is discarded), while the opposite does not. This knowledge is indicated in the value of the *is_lossy* slots in this relationship. The complete set of relationships between the ATC and GTC SONTs is depicted in Figure 5.8.

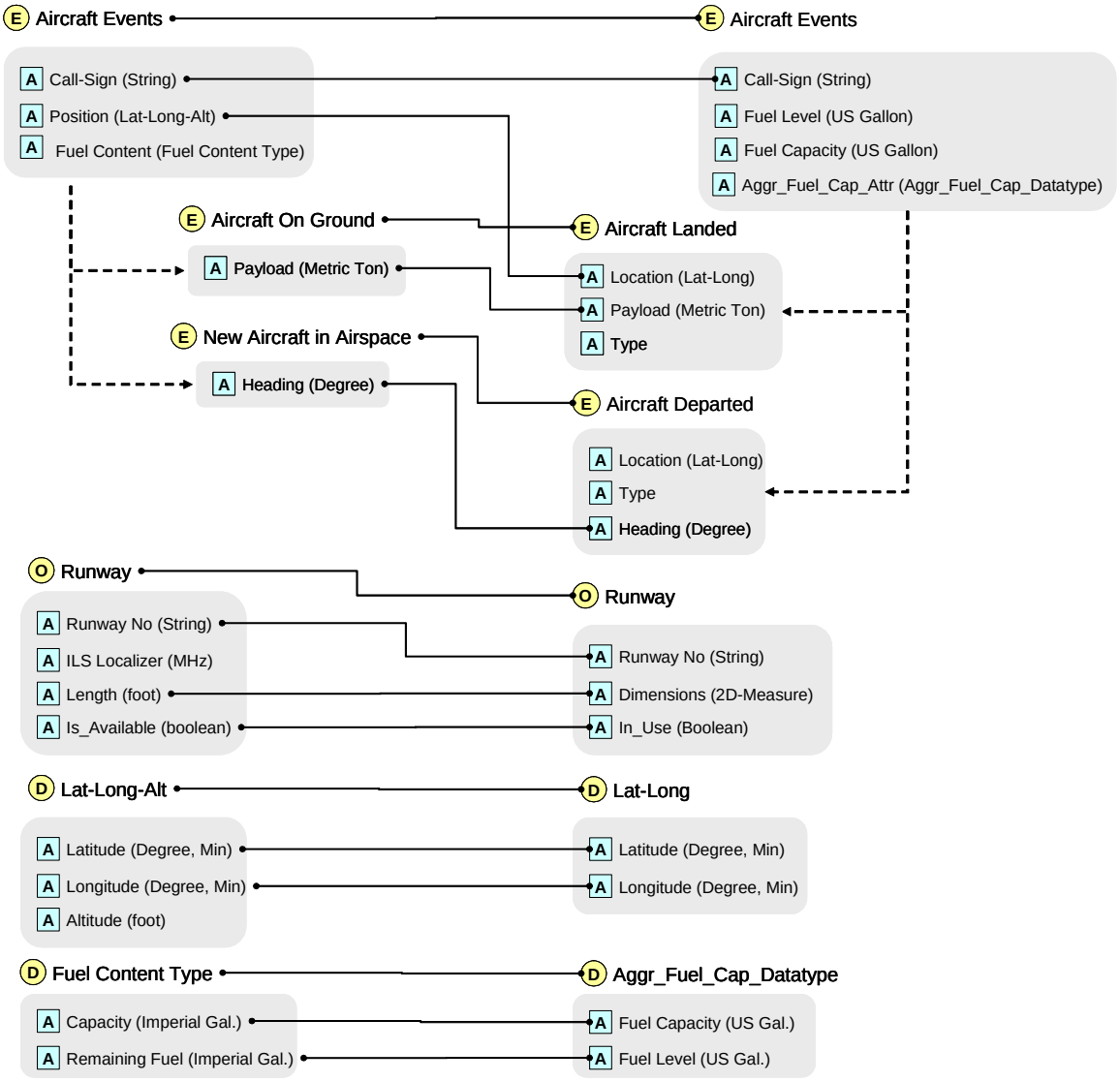


Figure 5.8: Relationships between ATC and GTC Entities

5.4.2 Relationships between GTC and Ground Services Entities

The information to be exchanged between the GTC and Ground Services federates relates primarily to activity at a gate. Therefore, all gate related objects and events in these SONTs are related to each other. Both the GTC and Ground Services SONTs model Gate

objects in their respective domain, between which a relationship instance is defined. The individual attributes of these gate objects that map to each other are *Gate Number*, *Sector* and *In_Use* in the GTC, and *Gate Number*, *Sector* and *Status* in the Ground Services SONT. The relationship between the *Gate Number* and *Sector* attributes is simply one of equivalence (hence the transformation does not need to be specified explicitly) and there are no lossy transformations involved. Since the attribute *Status* is an enumerated type, the transformations in the relationship between *Status* and *In_Use* must be defined explicitly (as was done with *Aircraft Type* and *Type* in the previous section).

Since the occurrence of an *Aircraft Parked* event in the GTC must trigger the *Service Required* event in the Ground Services simulation, (and conversely, the *Service Completed* event must trigger the *Aircraft Chocks Off* event) a relationship between these events must be specified as well. When an aircraft arrives at a gate in the GTC, information about its fuel level, fuel capacity and payload need to be subscribed to by the Ground Services federate. Therefore, we relate the *Fuel Level*, *Fuel Capacity* and *Payload* attributes of the *Aircraft Parked* event to the *Fuel Required* and *Payload* attributes, respectively, of the *Service Required* event. Since Both *Fuel Level* and *Fuel Capacity* are simultaneously required to determine the required fuel at a gate, there is an 2:1 mapping between these attributes. Recall that an aggregate data type (*Aggr_Fuel_Cap_Datatype*) and attribute (*Aggr_Fuel_Cap_Attr*) have already been defined to group *Fuel Level* and *Fuel Capacity* together (Section 5.4.1). These aggregate entities are employed again to specify a relationship between *Fuel Capacity*, *Fuel Level*

and *Fuel Required*. In a similar fashion, relationships are defined for corresponding attributes of the *Aircraft Chocks Off* and *Service Completed* events Figure 5.9.

Finally, a relationship between the *Runway* objects defined in both SONTs is instantiated. Specifically, the *Runway Number*, *Dimensions* and state of runways must be exchanged between the two federates. The relationship between the two *Runway Number* attributes is one of simple equivalence and does not involve any loss of information. The *Dimensions* attributes are conceptually equivalent, but they have different data types (*2D_Measurement* and *Length_x_Width*). Therefore, a relationship must be defined between these data types as well. Note that the attributes of *2D_Measurement* (*Length* and *Width*) correspond directly to the *Length* and *Width* attributes of *Length_x_Width*. Hence, the transformations in this data type level relationship do not need to be specified explicitly; it suffices to specify relationship between the individual attributes of *2D_Measurement* and *Length_x_Width* Figure 5.9. The relationship between the runway attributes Status and In_Use has already been defined above, so no additional steps need to be carried out. The full set of relationships between the GTC and Ground Services SONT entities is presented below in Figure 5.9.

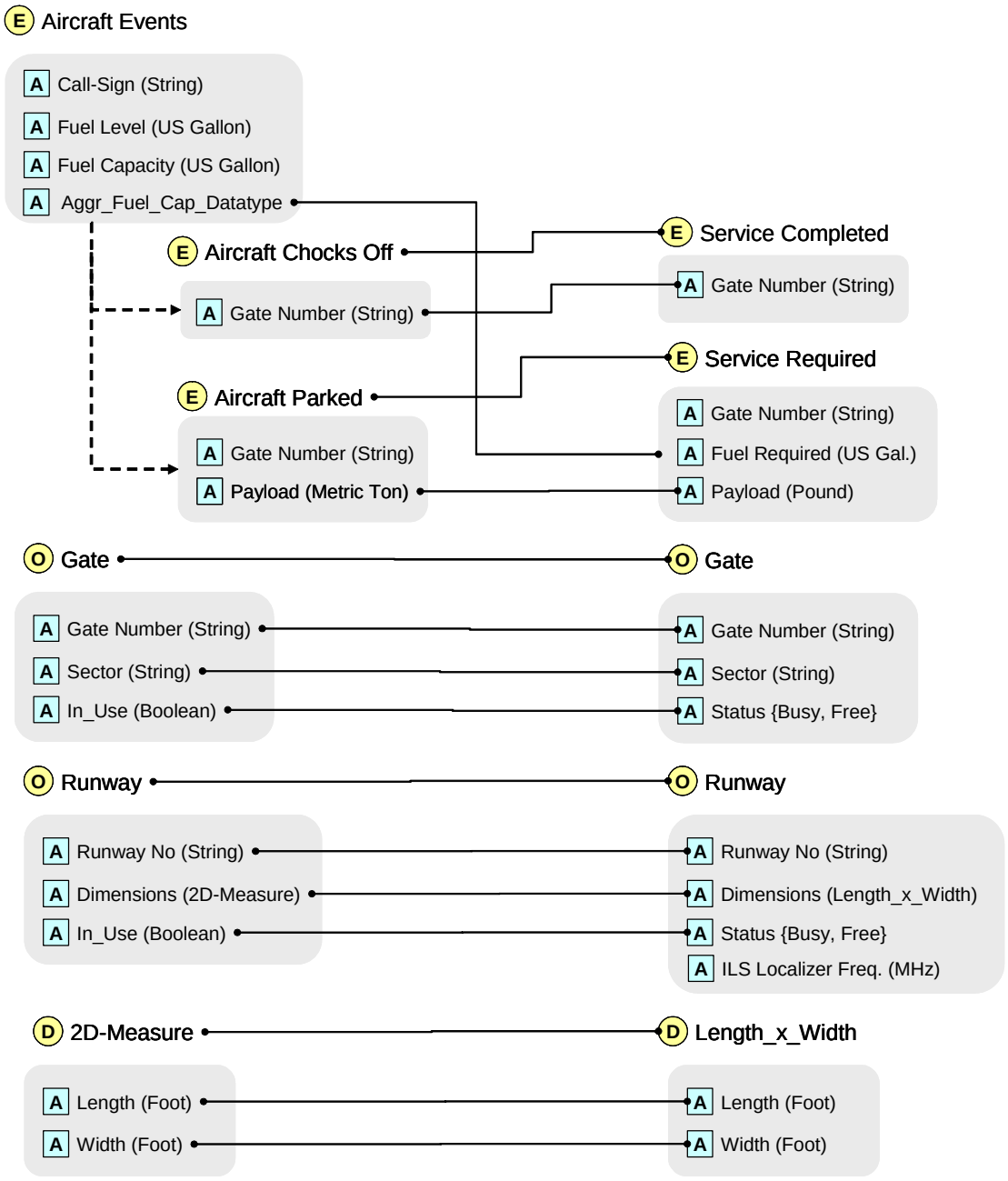


Figure 5.9: Relationships between GTC and Ground Services Entities

5.4.3 Relationships between the ATC and Ground Services Entities

The only information shared directly between the ATC and Ground Services federates is that of runways. In fact, the runway concept is modeled in each federate, and all federates exchange runway related information with each other. A relationship is defined between the *Runway* objects in both SONTs, following their individual attributes are related as well. The attributes *Runway Number*, *ILS Localizer*, *Length* and *Is_Available* in the ATC domain are related to attributes *Runway Number*, *ILS Localizer Frequency*, *Dimensions* and *Status*, respectively. One should keep in mind that most of these attribute level relationships no not really need to be instantiated. We have already related several attributes of the *Runway* object in the ATC to corresponding attributes in the GTC. In turn, these GTC attributes have been related to Runway attributes in the Ground Services domain. Based on these existing relationships, the relationship between two attributes of *Runway* objects in the ATC and Ground Services SONTs can be inferred by the GRIT algorithm. Therefore, we skip the specification of several attribute level relationships between the ATC and Ground Service federates. The only attribute-level relationship that does need to be instantiated is between the ILS localizer frequencies modeled in both federates. Recall that the GTC *Runway* object does not include any information about ILS. Hence, no prior relationship exists from which the relationship between the ILS localizer attributes of the ATC and Ground Services SONTs can be inferred. Therefore, we specify a relationship from *ILS Localizer to ILS Localizer Frequency* with no lossy transformations. Since their data types are the same, no further relationships or transformations need to be specified.

Using the knowledge captured in the relationships defined above, the GRIT algorithm is to be employed to produce a common schema for the set of related entities in the air traffic FONT. Furthermore, a set of 'final' SONT-Common entity relationships and transformations are to be specified in the FONT. Finally, the entities, relationships and transformations stubs captured in the FONT can be applied by a run time infrastructure to facilitate interoperation between the simultaneously executing federation of simulations. The application of the GRIT algorithm to complete the specification of this example FONT is elaborated in the next section.

5.5 FONT Generation

Having captured the relationships between SONT entities in the air traffic federation, we continue with the next and final step in the ontology-based framework process model—the generation of a common representation of shared entities and the transformation stubs relating them. Here, the knowledge captured in the relationships defined in the previous section is applied by the GRIT algorithm to perform the above mentioned tasks automatically. It is at this stage of the FONT development process that the fruit of all labor performed in earlier steps is enjoyed. By selecting a common representation leading to the least number of lossy transformations, and subsequently defining these transformations automatically, significant effort and time that would be required to perform these tasks manually is saved. In the following sub-sections, we trace through the execution of the GRIT algorithm in the context of the air traffic FONT developed thus far.

5.5.1 Common Representation Generation

The GRIT algorithm has its own process model (depicted in Figure 4.1), which we will follow to develop the common schema and transformations in the air traffic FONT. The first step of course, is to create a graph from the set of entities and relationships between them that have already been specified. Graphs are created for objects and events, their attributes and the attributes of related custom data types. To instantiate a common representation for all shared attributes of objects and events, an attribute graph is created such that each vertex corresponds to a single SONT attribute, and each edge corresponds to one of two transformations an attribute-level relationship. The result is a forest of many sub-graphs, wherein each sub-graph represents a set of vertices (attributes) that share a common representation. This forest is illustrated in Figure 5.10. Each vertex in this figure is identified by a number, which corresponds to an index number in the vertex array. Similarly each edge in this figure has a number which indicates its index in the edge array. Next to an edge label, the length of that edge (1 indicates a non-lossy transformation, 2*m (where m is the number of edges in the graph) indicates a lossy transformation) is included in parentheses. While we do not list the entire vertex or edge arrays (the equivalent information is presented in Figure 5.10), the attributes corresponding to each vertex in Figure 5.10 are listed in Table 5.2. As seen in this table, all attributes of the GTC, ATC and Ground Services that participate in a relationship, as defined in the previous section, are represented as vertices. Once the graph is initialized, Dijkstra's algorithm is invoked to identify the shortest paths between all vertices. The execution of this algorithm has been discussed in detail in Chapter 4; hence we do not explain it in detail in the context of this specific graph. Furthermore, the shortest path and

shortest distance lists associated with each vertex are not explicitly listed since these are evident from the figure below.

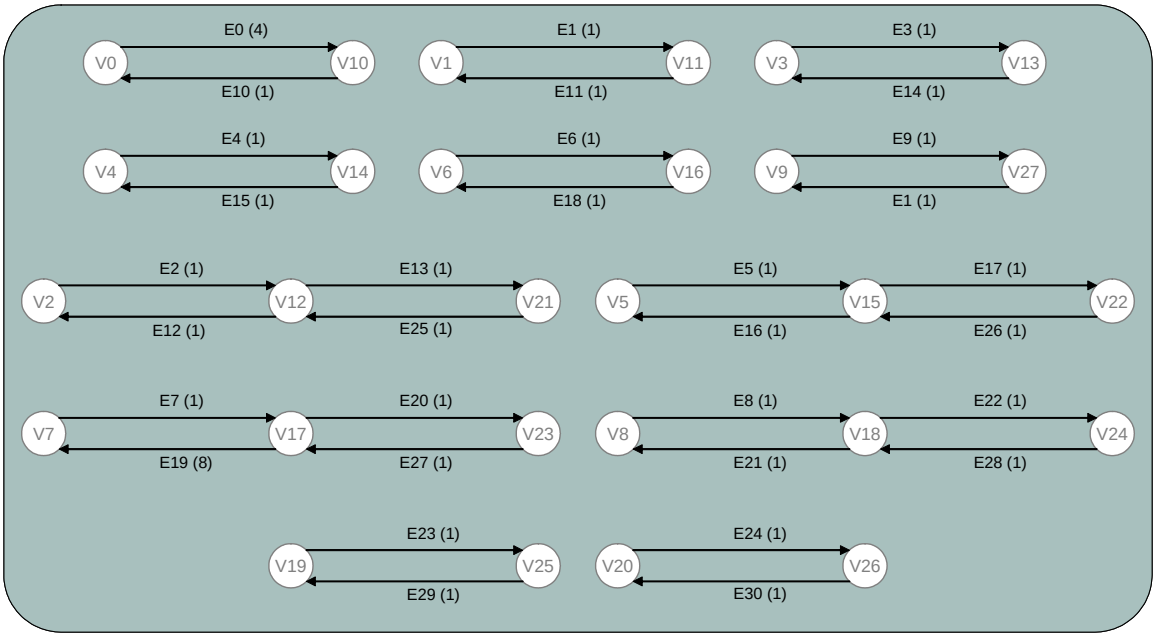


Figure 5.10: Air Traffic FONT Object and Event Attribute Forest

Having initiated the attribute graph and identified shortest paths between vertices, the procedure to select a common representation from a set of vertices in a sub-graph is executed. Here, each vertex (attribute) is hypothetically selected as the common representation, and the corresponding cost associated with that selection is captured. Recall that this cost is the total distance of the paths between each related pair of vertices, such that these paths pass through the selected common vertex. The vertex with the lowest cost associated with it, i.e. the lowest number of end-to-end lossy transformations,

is selected as the common representation. If two or more vertices have the same lowest cost, any of their associated attributes is selected as common.

Table 5.2: Air Traffic FONT Object and Event Attribute Vertex Array

Vertex	Attribute	Vertex	Attribute	Vertex	Attribute
V0	Position	V10	Location	V20	GTC Sector
V1	ATC Heading	V11	GTC Heading	V21	GS Payload
V2	ATC Payload	V12	GTC Payload	V22	Fuel Required
V3	ATC Call Sign	V13	GTC Call Sign	V23	GS Dimensions
V4	Aircraft Type	V14	Type	V24	GS Gate No
V5	Fuel Content	V15	Aggr_Fuel_Cap_Attr	V25	GS Sector
V6	ATC Runway No	V16	GTC Runway No	V26	Status
V7	Length	V17	GTC Dimensions	V27	ILS Localizer Freq.
V8	Is_Available	V18	In_Use		
V9	ILS Localizer	V19	GTC Gate No		

Most of the sub-graphs in the attribute forest only involve two vertices. The selection of the common representation in such cases is inconsequential. For example, consider the sub-graph involving vertices V0 and V10 i.e. a relationship between *Position* and *Location*. According to the GRIT algorithm, for a given vertex V_K selected as common, the associated cost is the sum of the shortest lengths from V_M to V_K and V_K to V_N, for

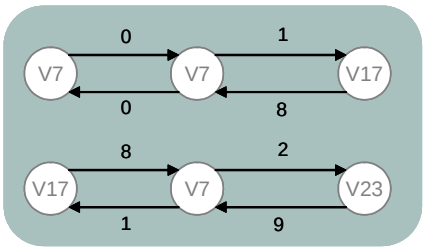
every related pair of vertices (V_M, V_N) in the sub-graph. If V0 is selected to be common, the path from V0 to V10 (passing through V0) has a minimal length of 4 (the weight of the lossy transformation is twice the number of edges, i.e., 2*2). Similarly, a path from V10 to V0, such that it passes through the common representation V0 has length 1. Therefore, the cost associated with V0 being common is 5. If V10 is selected to be common, the length of the path from V10 to V0, passing through the common vertex (V10) has length 1, while the length of V0—Common (V10)—V10 is 4. Therefore, the cost of selecting either attribute as common is the same. In other words, if *Position* is selected to be equivalent to the common representation, the transformation *Position*—Common—*Location* is just as lossy as the transformation from *Position* to *Location.* Similarly, the transformation *Location*—Common—*Position* is just as lossy as the transformation from *Location* to *Position.* In general, when there are only two vertices in a sub-graph, the cost of either one being common is the same. Therefore, the attribute corresponding to either vertex can be selected to be equivalent to the common representation shared between the two SONT attributes. In this example, we assume that *Position* is selected to be the common representation. Having done so, a new instance of the attribute metaslot, *Common_Position* (named by appending the SONT selected SONT representation to the word 'common') is created automatically, such that its data type is the same as *Position*'s (*Lat-Long-Alt*). Once the common attribute is instantiated, relationships between *Position* and *Common_Position* and *Location* and *Common_Position* are also specified automatically according to the GRIT algorithm. In this manner, the common representation for a pair of related SONT attribute, and relationships between the SONT and common attributes are captured in the FONT

automatically. Of course, there is still the issue of defining the transformation routines for these relationships, which we will explore shortly.

In sub-graphs involving three or more vertices, the cost associated with each vertex being selected as common can vary. That is, the selection of a given SONT attribute as the common representation does not always lead to the same number of lossy end-to-end transformations. As an example, consider the sub-graph involving vertices V7, V17 and V23 i.e. the attributes *Length*, *GTC Dimensions* and *GS Dimensions*. The cost associated with each of these attributes being selected as common is illustrated in Figure 5.11. Since the attributes *GTC Dimensions* and *GS Dimensions* both capture information about a runway's length and width, a transformation from any of these attributes to *Length* involves some loss of information. If *Length* is selected to be the common representation, a transformation from *GTC Dimension* to *GS Dimension* (and vice-versa) involves loss of information since this transformation has to be specified via the common representation (*GTC Dimensions*—Common (*Length*)—*GS Dimensions*). Obviously, this information loss is avoidable if *Length* is not selected to be common. Therefore, as is expected, *Length* has the largest cost associated with it. It is interesting to note that while the selection of *GS Dimensions* and *GTC Dimensions* leads to the same number of lossy end-to-end transformations, *GTC Dimensions* has a lower cost associated with it. This is because the selection of *GS Dimension* as the common representation means that a transformation from *Length* to *GTC Dimensions* (and vice-versa) has to be defined via *GS Dimensions*. Since there is no relationship specified by the user between *GS Dimensions* and *Length*, this relationship (and its transformations) has to be composed

based on the existing relationships between *Length* & *GTC Dimensions*, and *GTC Dimensions* & *GS Dimensions*. When *GTC Dimensions* is selected to be common, no transformations need to be composed from existing ones. Therefore, *GTC Dimensions* has the lowest cost in this sub-graph, and is selected to be equivalent to the common representation. Again, once a common attribute (*Common_GTC_Dimensions*) is defined, a set of SONT-Common relationships from *Length, GTC Dimensions* and *GS Dimensions* to *Common_GTC_Dimensions* are also instantiated.

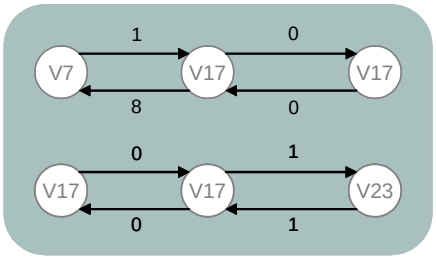
(a) V7 i.e. Length = Common



Cost= 29

The end-to-end transformation from GS Dimension to GTC Dimension is lossy, which is avoidable

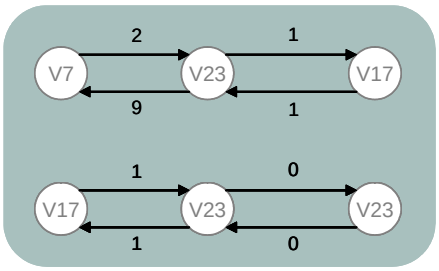
(b) V17 i.e. GTC Dimension = Common



Cost= 11

There is only one lossy transformation, which is unavoidable. All paths to and from the common attribute involve only one edge

(c) V23 i.e. GS Dimension = Common



Cost= 15

There is only one lossy transformation, which is unavoidable, but paths to and from the common attribute involve multiple edges.

Figure 5.11: Cost associated with Length, GTC Dimension and GS Dimension being selected as the common representation

Apart from the sub-graphs discussed above, most other sub-graphs do not involve any lossy transformations. Therefore, the cost associated with selecting any given vertex as common remains the same for all vertices in these sub-graphs. Consequentially, the GRIT algorithm may select any vertex in a sub graph to be equivalent to the common representation. The selection of the common representation in these cases is relatively

unimportant, hence we do not list which attributes are selected as common in these sub-graphs. Suffice it to mention that a common attribute is instantiated for all sub-graphs in the attribute forest, and the corresponding SONT-Common relationship instances are defined as well. The common representation for the attributes corresponding to each sub-graph in the attribute forest is illustrated below in Figure 5.12.

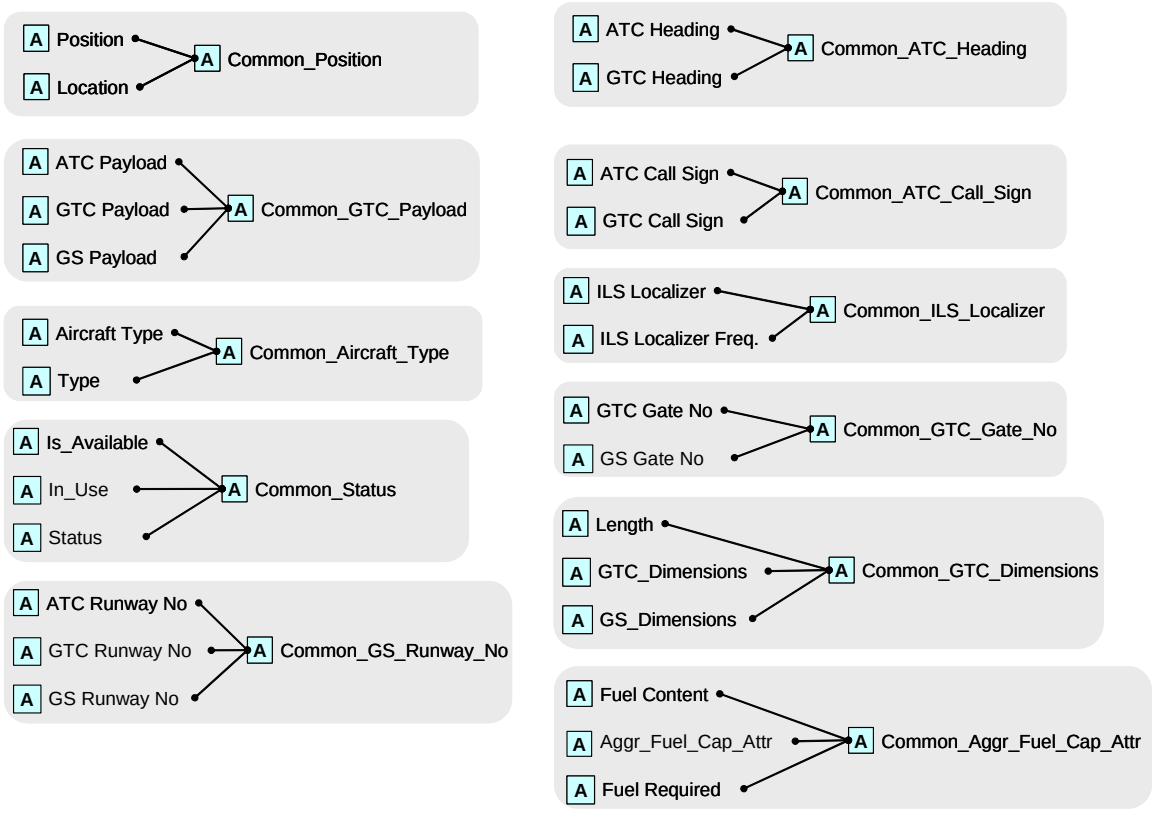


Figure 5.12: Common Representation and SONT-Common Relationships for all Object and Event Attributes in the Air Traffic FONT.

Once the common representation for all related attributes in the air traffic FONT have been generated, the next procedure invoked in the GRIT algorithm defines common

representations at the object and event level. Similar to the attribute graph defined above, an object and event level graph is defined, where vertices represent related objects and events, and edges correspond to the relationships between them. The object and event level graph for the air traffic FONT is illustrated in Figure 5.13. The entities corresponding to each vertex in this graph are listed in Table 5.3. The procedure to generate common objects and events begins by identifying sub-graphs with the overall forest. For each sub-graph, the attributes of the object or event corresponding to a given vertex are queried to determine if they are related to a common attribute. Each common attribute identified is then added to the domain of a new object or event class. Once all attributes of all vertices in the sub-graph have been traversed, the definition of this new, common class is complete. Furthermore, relationships are instantiated between the classes corresponding to each vertex in the sub-graph, and the newly created common class.

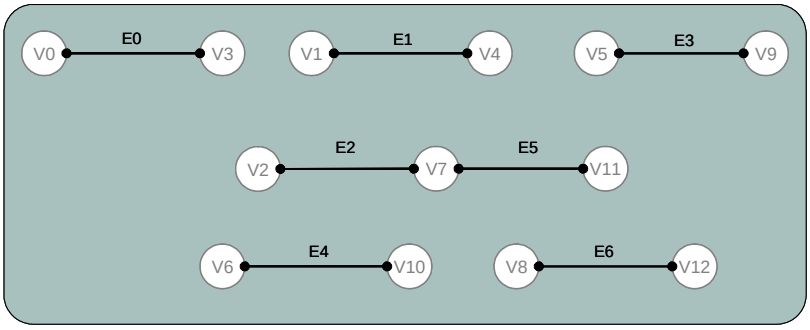


Figure 5.13: Air Traffic FONT Object and Event Forest

Table 5.3: Air Traffic FONT Object and Event Vertex Array

Vertex	Object / Event	Vertex	Object / Event	Vertex	Object / Event
V0	Aircraft On Ground (E)	V5	Aircraft Parked (E)	V10	Service Completed (E)
V1	New Aircraft in Airspace (E)	V6	Aircraft Chocks Off (E)	V11	GS Runway (O)
V2	ATC Runway (O)	V7	GTC Runway (O)	V12	GS Gate (O)
V3	Aircraft Landed (E)	V8	GTC Gate (O)		
V4	Aircraft Departed (E)	V9	Service Required (E)		

Let us examine the definition of a common object for the sub-graph with vertices V2, V7 and V11 in Figure 5.13 i.e. the *Runway* objects in each SONT domain. A new class, *Common_Object_1* is instantiated as the common representation for all three runway objects in the FONT. To determine the common attributes that comprise this new class, the attributes of the objects corresponding to each vertex in the sub-graph are traversed. For vertex V2 (the *ATC Runway* object), the set of SONT attributes related to attributes in the common domain are *ATC Runway No*, *Length*, *ILS Localizer* and *Is_Available* (Figure 5.3). We know that *Length* is related to *Common_GTC_Dimensions*, as was elaborated previously. Similarly, *ATC Runway No* is related to *Common_GS_Runway_No*, *ILS Localizer* is related to *Common_ILS_Localizer*, and *Is_Available* is related to *Common_Status* (Figure 5.12). These common attributes are modified such that their domain includes the new class *Common_Object_1*. Furthermore, a relationship instance is defined from *ATC Runway* to *Common_Object_1*. This process is then repeated for vertices V7 and V11. Since the attributes of *GTC Runway* and *GS*

Runway are also related to the same common attributes as those of V2, no further attributes are added to *Common_Object_1*'s domain. (In general, if multiple attributes are related to the same common attribute, the addition of that common attribute to the target domain is performed more than once. This does not mean that we end up with too many common attributes describing the common class. Rather, it means that the fact that a unique attribute describes a unique object or event is specified several times). In this manner, a common class (object or event) and a set of SONT-Common relationships is specified for each sub-graph in the forest depicted in Figure 5.13.

Having defined the common representation for all shared entities in the air traffic FONT, we proceed to the final automated step in FONT development; the derivation of transformation. For every SONT-Common attribute level relationship (match) that has been specified in this section, a corresponding mapping must be instantiated. We explore the instantiation of these mappings, as accomplished by the GRIT algorithm, in the following section.

5.5.2 Transformation Stub Generation

For every relationship between a SONT domain attribute and a common attribute, a corresponding pair of transformation stubs must be defined. The GRIT algorithm invokes a procedure to do so after the SONT-Common relationships for a given sub-graph of the attribute forest have been instantiated. For each relationship from a SONT attribute to a common attribute, the corresponding shortest path between the two is identified (equivalently, the shortest path between the given SONT attribute and the attribute that

was selected to be the common representation), based on which the required transformations are generated. Each edge in this shortest path implies that another transformation has to be added to the chain of routines composed together to realize the required SONT-Common transformation. Of course, for the SONT attribute previously selected to be the common representation, a shortest path to the common attribute does not explicitly exist. The transformation between the two is a simple equivalence operation, which is generated by the GRIT algorithm, given it keeps track of which attribute is selected as common in a given sub-graph.

Let us explore the instantiation of transformations in the sub-graph of the attribute tree consisting of vertices V0 and V10 i.e. attributes *Position* and *Location* (Figure 5.10). The instantiation of the corresponding common attribute *Common_Position*, such that its representation is equivalent to *Position*, has been elaborated in Section 5.5.1. A relationship between *Location* and *Common_Position* has also been defined, for which the transformations are now specified. The value of the *function_to* slot of this relationship is entered by the GRIT algorithm based on the shortest path from *Location* (V10) to the attribute select to be equivalent to *Common_Position*, i.e. *Position* (V0). From Figure 5.10, it is clear that this path involves a single edge E10. Since a transformation corresponding to this edge has not been explicitly defined, the GRIT algorithm assumes that the transformation from *Location* to *Position* only involves a conversion between their data types (*Lat-Long* and *Lat-Long-Alt*, respectively). Therefore, the required transformation is specified as:

```
Lat-Long-Alt Location_to_Common_Position (Lat-Long input)
   {
      Lat-Long-Alt output;
      output = Lat-Long_to_Lat-Long-Alt (input);
      return output;
   }
```

Note that this transformation is defined in terms of the transformation between the data types of *Location* and *Position*. While this data type level transformation has not yet been generated, since we have previously specified the existence of a relationship between *Lat-Long* and *Lat-Long-Alt*, the required transformation will be generated before the FONT specification is complete. Also, since the path from *Location* to the common representation (*Position*) only involves a single edge, only one transformation (*Lat-Long_to_Lat-Long-Alt*) is called from within the transformation from *Location* to *Common_Position*. In a similar fashion, the transformation from *Common_Position* to Location i.e. the value of *function_from* in the relationship between *Location* and *Common_Position* is generated as:

```
Lat-Long Common_Position_to_Location (Lat-Long-Alt input)
   {
      Lat-Long output;
      output = Lat-Long-Alt_to_Lat-Long (input);
      return output;
   }
```

The only other SONT attribute in this sub-graph is *Position*. Since *Position* corresponds to the vertex that was selected as the common representation, the shortest path from

Position to itself is not investigated. Instead, the required *function_to* transformation in the relationship from *Position* to *Common_Position* is simply specified as listed below (the required *function_from* transformation stub is also specified as an equivalence relationship).

```
Lat-Long-Alt Position_to_Common_Position (Lat-Long-Alt input)
   {
      Lat-Long-Alt output;
      output = input;
      return output;
   }
```

The generation of transformation stubs for all other sub-graphs in the attribute forest follows the same process. While we do not explain the generation of each transformation in detail, it is worth briefly studying the generation of a transformation from the attribute *Type* (V14) to its corresponding common representation *Common_Aircraft_Type*, which is defined so as to be equivalent to the attribute *Aircraft Type* (V4). Recall that we explicitly defined a transformation from *Type* to *Aircraft Type* (and vice-versa) in Section 5.4.1. This means that the transformation between *Type* and *Common_Aircraft_Type* cannot be defined solely in terms of a transformation between their data types. Instead, the GRIT algorithm identifies that a user-specified transformation exists corresponding to the edge connecting V14 to V4, and generates the required SONT-Common transformation in terms of this user-defined transformation:

```
String Type_to_Common_Aircraft_Type (String input)
   {
      String output;
      output = Type_to_Airraft_Type (input);
      return output;
   }
```

Once the GRIT algorithm has generated all required transformations for all sub-graphs of the attribute forest, the transformations between their respective data types are to be specified. These relationships include those between primitive data types, such as two units of measurement, and those involving custom data types, such as *Lat-Long* and *Lat-Long-Alt*. For every related set of data type classes, an appropriate procedure is invoked to generate the transformation between them. For two related primitive data types, the generation of the required transformations stub is relatively simple. The knowledge as to how two primitive data types relate is captured when the World Ontology is defined. For example, the relationship between two units of measurement is captured in terms of their conversion factor to a selected reference unit. If two custom data types relate such that their individual attributes map to each other, and the relationship between these attributes is that of equivalence, other than their representation (data types), the GRIT algorithm can generate transformations for these related data types automatically. All other data type level relationships must be specified explicitly.

Since the *GTC Payload* attribute is expressed in metric tons, and the Ground Services *Payload* attribute has data type *Pound*, a relationship between these to units of measure is specified, for which transformations are to be generated. Recall that the definition of each

unit data type class in the World Ontology includes the slot *SI_Conversion_Factor*, which represents the factor by which a value in a given unit has to be multiplied to achieve the equivalent value in SI Units (refer to Section 3.7.2). The Data type class Metric *Ton* has a conversion factor of 1016, while *Pound* has a conversion factor of 0.45. Using these values, the GRIT algorithm generates the required transformation from *Metric Ton* to *Pound* as:

```
Float Metric_Ton_to_Pound (Float input)
   {
      Float output;
      output = (input * 1016) /0.45;
      return output;
   }
```

A relationship between two custom data types in the air traffic FONT exists between *Lat-Long* and *Lat-Long-Alt*, which are the data types of the attributes *Position* and *Location*, respectively. The individual attributes of these data types that relate to each other are *ATC Latitude* & *GTC Latitude*, and *ATC Longitude* & *GTC Longitude*. Since these attributes are conceptually equivalent, the transformations between *Lat-Long* and *Lat-Long-Alt* can be derived automatically by the GRIT algorithm, given that relationships are defined between *ATC Latitude* & *GTC Latitude*, and *ATC Longitude* & *GTC Longitude*. Recall that we have defined these relationships in Section 5.4.1. In order to generate the required data type transformation, the GRIT algorithm maintains a graph of all related data type attributes. From this graph, the shortest path (if it exists) between two attributes belonging to the respective data types being related is identified, based on

which a conversion from one attribute to the other is generated. This process is repeated for all related attributes of the two data types.

The graph of all data type attributes in the air traffic FONT is illustrated in Figure 5.14, and the attributes associated with the different vertices are listed in Table 5.4. In this graph, the vertices corresponding to attributes of *Lat-Long-Alt* are V0, V1, and V2 (*ATC Latitude, ATC Longitude* and *Altitude*). There exists a path between from V0 to V5, and V1 to V6, where V5 and V6 correspond to *GTC Latitude* and *GTC Longitude,* which are attributes of Lat-Long (note that no attribute of *Lat-Long* is reachable from the ATC attribute *Altitude* (V2)). The GRIT algorithm defines a transformation from *ATC Latitude* to *GTC Latitude* as a composition of all edges in the shortest path from V0 to V5. Since this path involves only one edge, and the data types of both *ATC Latitude* and *GTC Latitude* are the same (the unit data type *Degree-Minute*), this transformation is reduced to a simple equivalence operation. Similarly, a transformation from *ATC Longitude* to *GTC Longitude* is also generated in a similar fashion. At this point, all attributes of *Lat-Long-Alt* that have paths to any attributes of *Lat-Long* have been accounted for. The required data type level transformation is specified as a collection of these attribute conversion, as follows:

```
Lat-Long Lat-Long-Alt_to_Lat-Long (Lat-Long-Alt input)
   {
      Lat-Long output;
      output.GTC_Latitude = input.ATC_Latitude;
      output.GTC_Longitude = input.ATC_Longitude;
      return output;
   }
```

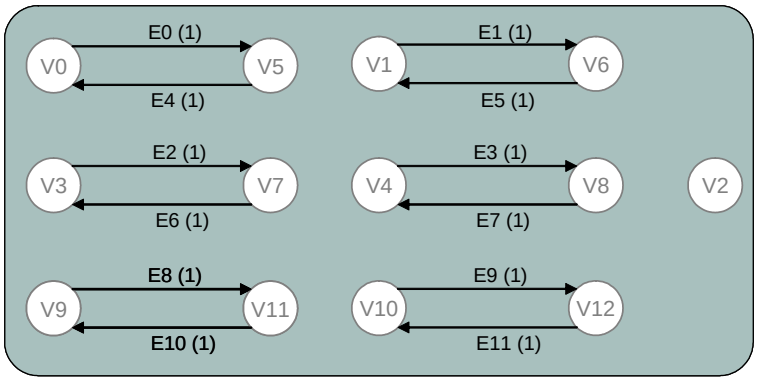


Figure 5.14: Air Traffic FONT Data Type Attribute Forest

Table 5.4: Air Traffic FONT Data Type Attribute Vertex Array

Vertex	Attribute	Vertex	Attribute	Vertex	Attribute
V0	ATC Latitude	V5	GTC Latitude	V10	Width
V1	ATC Longitude	V6	GTC Longitude	V11	GS Length
V2	ATC Altitude	V7	Fuel Level	V12	Breath
V3	Remaining Fuel	V8	Fuel Capacity		
V4	Capacity	V9	GTC Length		

Another example of a custom data type transformations generated by the GRIT algorithm is that of the relationship between the data types *Fuel Content Type* and *Aggr_Fuel_Cap_Datatype*. The attributes of *Fuel Content Type,* namely *Remaining Fuel* and *Capacity*, are represented in the graph above as vertices V3 and V4. A path exists from V3 to V7 and V4 to V8, where V7 and V8 correspond to attributes *Fuel Level* and *Fuel Capacity* in the domain of *Aggr_Fuel_Cap_Datatype*. Since these attributes are equivalent to each other, the conversions between them can be generated based on the relationship between their data types. Note that *Remaining Fuel* and *Capacity* have data type *Imperial Gallon*, and *Fuel Level* and *Fuel capacity* are expressed in *US Gallon*. Therefore, the transformation from *Capacity* to *Fuel Capacity* involves a transformation from the primitive unit data type *Imperial Gallon*, to *US Gallon* (the same is true for the transformation from *Remaining Fuel* to *Fuel Level*). The resultant data type level transformation from *Fuel Content Type* to *Aggr_Fuel_Cap_Datatype* then automatically generated as a collection of these attribute transformations:

```
Aggr_Fuel_Cap_Datatype
Fuel_Content_Type_to_Aggr_Fuel_Cap_Datatype (Fuel_Content_Type
input)
   {
      Aggr_Fuel_Cap_Datatype output;
      output.Fuel_Capacity = Imperial_Gallon_to_US_Gallon
      (input.Capacity);
      output.Fuel_Level= Imperial_Gallon_to_US_Gallon
      (input.Remaining_Fuel);
      return output;
   }
```

There exist relationships involving custom data types in the air traffic FONT, for which the GRIT algorithm cannot derive transformations. An example that has already been discussed is the transformation between two enumerated data types. Recall that both attributes *Aircraft Type* (in ATC domain) and *Type* (in GTC domain) have enumerated data types, and the transformation between them has been explicitly specified. This is because enumerated data types do not have individual attributes that can be related to each other. If *Aircraft Type* and *Type* were modeled in terms of data types with individual attributes, then a relationship between these attributes could be specified. The knowledge captured within these relationships could then be used by the GRIT algorithm to compute the required data type level transformation.

The same is true for any relationship between a custom data type and a primitive data type. Most primitive data types do not have individual attributes. The only primitive data types that have slots (note slots in general, not instances of the attribute metaslot) are unit data types, but their slots capture information about the conversion factor to an SI unit. Therefore, no relationships can be specified between the attributes of a custom data type and those of a primitive data type, because the latter is non-existent. Hence in relationships between attributes where one attribute has a primitive data type, and the other has a custom data type, the associated transformations must be explicitly provided, since they cannot be derived based on a data type level transformation that does not exist. This is precisely the reason why the transformation from attribute *Length* to *GTC Dimension* (and vice versa) was explicitly defined in Section 5.4.1.

Given that all transformations that cannot be generated by the GRIT algorithm are explicitly defined, once the procedure to generate data type transformations completes its execution, the execution of the overall GRIT algorithm is complete. The result is a complete FONT specification, wherein a common set of objects, event and attributes has been instantiated as the communal representation for all shared federate entities. Furthermore, a set of transformation stubs to convert all airport related concepts from their respective SONT representations to their related common representation, and vice-versa are captured in the FONT. The entire set of information contained within the FONT can then be applied by an RTI to facilitate consistent run-time communication between the ATC, GTC and Ground Services simulations.

Based on the nature of the automatically derived transformations between SONT and Common entities (whether they are lossy or not), we may choose to specify additional relationships and re-iterate through the GRIT algorithm. There are only two lossy SONT-Common relationships (From *Common_Position* to *Location* and from *Common_GTC_Dimensions* to *Length)* that have been derived. Since the relationship between *Location* and *Position* (equivalent to *Common_Position*) or *Length* and *GTC Dimension* (equivalent to *Common_GTC_Dimensions*) inherently involves discarding some information, we cannot specify any additional knowledge to help the situation. Furthermore, the information that is discarded is inconsequential to the interplay between federate simulations. Hence, we accept this specification of the FONT as the 'final' specification to be used by the RTI.

5.6 Empirical Performance Validation

In the previous sections, we have stepped through the process of developing a FONT for the example air traffic federated simulation. A system level simulation of the airport being designed was to be developed and executed. Sub-system level simulation models exist and need to interact with each other in order to simulate the emergent behavior of the airport as a whole. The run-time exchange of information between distributed simulations requires a common information model that defines the (representation of) objects and events that simulators can publish or subscribe to. However, several concepts that are common to the airport as a whole (e.g. runway, aircraft) are modeled in different ways in each sub-system level simulation. An archetypical example of this is that the concept of an aircraft's payload is represented in metric tons in the ATC and GTC simulation models, and in pounds in the Ground Services model. We used the ontology-based framework to allow the three federates to exchange information in a consistent manner without having to modify each one of their implementation. Our goal in applying this framework is to determine a common information model and set of transformation stubs to and from that model in an automated fashion. Using these, a run-time infrastructure can facilitate the execution of the required system-level simulation such that information between the GTC, ATC and Ground Services federates is exchanged in a consistent manner, and thereby the expected behavior of the airport is correctly simulated.

To that extent, we pose the question—"Has the ontology-based framework done what it was supposed to do?" In a broad sense, the answer is a resounding yes. Given the ATC,

GTC and Ground Services simulation models, with the knowledge of the representation of shared concepts within their individual domains and the loss of information in transforming between them; a common information model and a set of transformation stubs to convert information to and from that common representation have been generated automatically. That being said, we confess that the word ‘automatically’ is used rather loosely here. There was still some effort involved in modeling the individual simulation domains in ontologies, and specifying relationships between them. Moreover, not all transformations were generated automatically. Therefore, considerable time and effort is still required to achieve representational compatibility between the ATC, GTC and Ground Services simulations. However, this is significantly less time and effort intensive than having to define the common information model and transformations manually. In this federation, there are 26 shared attributes for which a common representation has defined and 56 corresponding transformation stubs that have generated automatically (Figure 5.10). This of course does not include transformations between the data types of these attributes. While we did not attain representational consistency in the air traffic federation in a completely automated fashion, clearly, the use of this framework has automated part of the process. Therefore, the framework has successfully been used to *support* automation in the process of attaining representational compatibility between the three federates in the air traffic simulation. Looking back at research question 1 and its associated hypothesis, the framework has accomplished exactly what it was supposed to do, in the context of this example.

At a finer level of granularity, it is important to evaluate the performance of the framework in terms of its key characteristics that we set out study. In Section 5.2, we stated that the goal of undertaking this example FONT development was to answer three performance-related questions about the framework, namely (i) Is the expressiveness of the world ontology sufficient to capture all SONT model concepts? (ii) Is the expressiveness of relationships sufficient to capture required knowledge about how two concepts relate to each other? and (iii) Does the GRIT algorithm determine the common representation and transformations in a correct and effective manner? As we shall see below, answering these questions helps to determine the validity of the research hypotheses we posed in Chapter 1, in the scope of this example problem.

5.6.1 Expressiveness of the World Ontology

Since the World Ontology has been based on the HLA OMT; we expect that different simulation concepts can be represented in terms of Objects, Events and their Attributes. In this example, we have modeled several different concepts using the constructs defined in the World Ontology, ranging from a static, persistent entity such as a runway, to the act of performing some action, such as re-fueling an aircraft. We were able to capture knowledge as to the persistence of a given concept in the simulation by either modeling it as an event or an object. By modeling objects and events as classes, significant freedom is offered in how each simulation concept is modeled. Each object and event can have any number of attributes, which means that pretty much any concept can be described in an object oriented fashion. This description can be captured completely in one attribute with a complex data type, or in several attributes whose data types are simpler in nature.

For example, we modeled the position of an aircraft to contain information about its latitude, longitude and altitude (in the *Lat-Long-Alt* data type) in the ATC (Figure 5.3). We could just as easily have specified the attributes *Latitude*, *Longitude* and *Altitude* as descriptors of an aircraft. We chose the former because it reflects how these concepts have been modeled in the underlying simulation model. The fact that we were able to represent these concepts in the exact same fashion as the simulation model shows that the World Ontology does not constrain the way in which simulation concepts are represented.

However, the fact that all concepts in a SONT or FONT must be expressed in an object oriented fashion is a limitation that implies additional work to attain representational consistency in the air traffic federation. Recall that the Ground Services simulation model is a legacy simulation model that does not employ an object oriented representation. Since the World Ontology constrains the representation of simulation models in terms of abstract data types (objects) and methods (events), an object oriented interface is to be built as a wrapper around the Ground Services federate.

Aside from imposing an object oriented representation, there are very few constraints imposed by the World Ontology. Whether a concept is described such that its properties are grouped together in a single attribute (as with Position, a complex descriptor of an aircraft) or specified as multiple, more simple attributes, the knowledge represented is equivalent. There is some added effort required in specifying complex data types, but this is almost negligible. In the end, the knowledge that needs to be provided as input to the

GRIT algorithm, and consequently the time and effort required to do so does not change if a concept is modeled with many simple attributes (with primitive data types) or few complex attributes (with custom data types).

It is worth mentioning that in the scope of this framework, objects and events are treated no differently, so it really didn't matter whether we decided to model a federate concept as an object or an event. For example, we chose to model the concept of an aircraft only as an event, even though aircraft are somewhat persistent in the ATC and GTC simulations. Knowing that the only time aircraft information is exchanged between the ATC and GTC is when an event occurs in relation to an aircraft, it made sense to model aircraft related concepts as events. If we had modeled them as objects, the underlying relationships and transformation stubs relating individual attributes would not change. The only change comes during execution, where a change in an aircraft's attributes in the ATC would be reflected in a corresponding GTC aircraft, irrespective of whether that aircraft is landing or not (vice-versa applies as well). The change in an aircraft objects attributes would then trigger a local event, as opposed to a direct mapping between events, as is currently implemented.

Based on the above arguments, and the fact that we were successfully able to model all three federates in terms of World Ontology constructs, we accept the validity of the World Ontology in performing as required.

5.6.2 Expressiveness of Relationships

As we have seen in previous sections, we were successfully able to specify the knowledge required to generate a complete FONT specification in multiple instances of the relationship class. This includes knowledge of matches between objects, events and data types (such as the relationship between *ATC Runway* and *GTC Runway*), and the mappings between their individual attributes (such as the relationship between *Location* and *Position*). By specifying relationships between objects (or events and data types) independently, the knowledge as to how two attributes relate does not have to be provided again and again if the attribute appears in multiple objects. For example, the relationship between attributes *In_Use* (in the domain of both *GTC Runway* and *GTC Gate*), and *Status* (seen in both *GS Runway* and *GS Gate*) only had to be specified once. Furthermore, we were able to specify simultaneous mappings from multiple attribute to one attribute, and vice versa through the definition of aggregation data types and attributes. In the relationship between *Fuel Content* and *Fuel Level & Fuel Capacity*, the aggregation of the latter two in a single attribute meant that we could still specify a transformation stub from *Fuel Content* to *Fuel Level* and *Fuel Capacity*, such that the procedure only has a single output parameter. In addition, the *is_lossy* slots with each relationship allowed us to capture knowledge as to the loss of information in translating between shared attributes, which is required to generate the required transformation stubs. Where these could not be inferred automatically, the *routine* slots allowed us to specify transformations explicitly.

A significant limitation in expressing relationships was observed in that several transformations needed to be specified manually. If the relationship between two attributes is any more complex than a conversion between their representations (data types) additional knowledge required as to the concept level relationship between two attributes cannot be specified; instead the transformations need to be specified explicitly. Effectively, this means specifying the same relationship twice (as two procedures, instead of one declarative relationship). Furthermore, transformations for relationships involving enumerated data types must be specified manually. As was experienced with the relationship between attributes *Aircraft Type* and *Type*, there is no provision to specify matches between the individual enumerals (such as *Turbo Prop* equates to *Propeller*). Instead, the entire transformation procedure is to be listed as a switch or if-else procedure (see Section 5.4.1). Moreover, this knowledge has to be provided twice, once in the procedure to convert the value of *Aircraft Type* to a corresponding value of *Type*, and once to go the other way. The same is true for the relationship between *In_Use* and *Status* (see Section 5.4.2). Similarly, for relationships involving one primitive data type and one custom data type, transformations need to be defined manually. In defining a relationship between data types *2D-Measurement* and *Foot*, the attributes of *2D-Measurement* could not be mapped to any corresponding attributes in *Foot*, simply because this, and all primitive data types don't have any attributes. Hence, the required transformation was specified manually.

While some transformations do need to be specified explicitly, it should be noted that since the majority of attributes in the air traffic FONT have primitive data types, the

relationships between them were defined in a relatively straightforward and quick fashion. For these relationships of course, we did not need to specify any transformations. Moreover, we did not have to specify all relationships; some could be inferred from others. Recall that the set of relationships defined between the ATC and GS SONT entities was minimal, in that most of their concepts were related through existing relationships with the GTC SONT (Section 5.4.3).

Clearly, the relationship class allowed us to specify all the knowledge required to complete the FONT specification. As we have noted, the efficiency in specifying this knowledge has room to improve based on the fact that several transformations had to be specified manually. However, the relationships in the FONT have 'done what they are supposed to do' to support the development of the air traffic FONT, which is our basis for accepting the performance validity of this component.

The two points we have evaluated thus far lend support to accepting the overall validity of Hypothesis 2 posed in Chapter 1, in that we have shown, with the help of a concrete example, that a metamodels (the World Ontology) can be used as a vocabulary for specifying SONTs, and that relationships between SONT entities can be defined based on those defined in this metamodel.

5.6.3 Correctness and Efficiency of the GRIT Algorithm

For all attribute sub-graphs in the air traffic FONT involving lossy transformations, the GRIT algorithm correctly selected a common representation leading to the least number

of end-to-end lossy transformations. We traced through the procedure to determine the common representation for the two sub-graphs involving lossy transformations, namely those corresponding to (i) the relationship between *Position* and *Location*, and (ii) the relationship between *Length*, *GTC Dimensions* and *GS Dimensions*. The latter highlights an important point about this procedure—even though the selection of *GTC Dimensions* and *GS Dimensions* as common leads to the same number of minimal lossy transformations, *GTC Dimensions* has a lower cost associated with it. This is because the cost associated with each attribute reflects not only the number of lossy transformations, but the number of chains in an end-to-end transformation as well. Therefore, in selecting the representation with lowest cost as common, the GRIT algorithm identifies that *GTC Dimensions*, if selected as common, leads to the least number of lossy transformations, and the most efficient composition of SONT-Common transformations (Figure 5.11). For all other sub-graphs in the attribute tree, there are no lossy transformations, so the selection of the common representation is somewhat inconsequential. Even then, the GRIT algorithm selected the common representation such that the simplest composition of SONT-Common transformations is realized.

The correctness of the transformation stubs generated is somewhat dependent on the knowledge provided as input to the GRIT algorithm. Under the assumption that all transformations that were generated automatically correspond to relationships where the same concept is defined with a different representation (data type), the transformations produced are exactly as desired. In the development of the air traffic FONT, all transformations associated with relationships between two disparate concepts were

specified manually. Hence, all transformations derived automatically (such as between *GTC Payload* and *GS Payload*) are exactly as desired. Another assumption is made in that for every set of related attributes, a relationship between their data types exists. The transformation from *GTC Payload* to *GS Payload* makes a call to a data type level transformation from *Metric Tons* to *Pounds*. If this relationship had not been defined before executing the GRIT algorithm, the resultant air traffic FONT would not be complete. That is, a run-time or compile-time error would occur when a call to a non-existent procedure is flagged. However, the occurrence of such a situation is more the fault of the federation developer and does not mean that the transformation generation procedure is flawed.

The efficiency of the GRIT algorithm in generating the required complete air traffic FONT can be quantified in terms of the overall complexity of executing its individual procedures. In Chapter 4, bounds on these complexities were defined in terms of the number of related attributes, objects, events and so on. In Table 5.5, we have quantified those bounds in the context of this example, given that we know the total number of vertices and edges in each graph. The total indicates that the number of instructions processed in completing the execution of the GRIT algorithm is $\leq \alpha * 23596$. Assuming that α, the number of instructions in the inner-most loop of each procedure, is not very large, it is evident that the GRIT algorithm is quite efficient, and does not require significant computing resources. Realize that today's computers can perform millions of instructions per second. That being said, the total number of instructions noted below is at

the application code level. Of course, these correspond to may more instructions in assembler or machine code, but this is still well within reason.

Table 5.5: Complexity of the GRIT Algorithm Execution in the Air Traffic FONT Example

Procedure	Theoretical Complexity	Empirical Complexity
Generate Shortest Path List	$O(N^2*m)$	22599
Select Attribute Common Representation	O (N*T*m)	208
Select Object Common Representation	O (S*T*a)	60
Generate Attribute Transformation Stub	$O (N^2)$	729
TOTAL		23596

The successful execution of the GRIT algorithm to automatically complete the air traffic FONT specification provides significant support to the validity of Hypotheses 3. In the context of this example, the GRIT algorithm automatically derives the common information model and infers transformations based on existing knowledge (semantics). The graph-based approach is found to be an efficient manner in which these tasks are undertaken. Essentially, this is exactly the vision proposed in hypotheses 3, which we have established is certainly valid within the scope of this example problem.

The goal of this chapter has been to gain insight as to the performance validity (usefulness) of the framework and algorithm developed previously. Based on the

arguments above, we have accepted the empirical performance validity of the research hypotheses. Now, we must generalize the performance validity acceptance beyond the scope of the air traffic federated simulation. The functionality and fundamental limitations of the framework, as applicable to a broad range of federated simulation scenarios are discussed in the following chapter. Based on this discussion, we make a 'leap of faith' to accept the general validity of this framework.

CHAPTER 6

EVALUATION AND REFLECTION

The purpose of this chapter is to bring closure to the development and analysis of the ontology-based framework presented in this thesis. In doing so, we consolidate the various ideas developed throughout this document and tie them back to the research questions and hypotheses posed at the outset. The idea here is to summarize and re-emphasize the contributions made in this body of research, and highlight its inherent limitations. In this manner, the reader is left with a clear understanding of the specific points that have been addressed in this thesis. A Masters Thesis grounded in research requires one to pose meaningful research questions, establish and develop associated hypotheses, and finally build confidence in their validity. In this chapter, we undertake the final step in this process, wherein we consolidate arguments in support of the general validity of all three hypotheses posed in Chapter 1. Finally, we close with a brief overview of potential paths along which future work can be undertaken based on what we have accomplished thus far. In answering the research questions, we have uncovered limitations to our hypotheses that lead to a new set of questions. In the interest of expanding the range of the usefulness of our contribution it is important that we discuss avenues for future investigation.

6.1 A Critical Review of this Research

Having conducted and explained the research contributions and applied them to an example problem in previous chapters, it is important to tie all of this back to the big picture—supporting the process of developing federated simulations. In this section, we re-emphasize how the individual constructs (SONTs, FONTs, graph-based algorithm and so on) factor into supporting automation and reuse in federation development. Furthermore, we highlight the fundamental limitations of our approach to achieving representational compatibility in a federated simulation. By doing so, we are able to bound or characterize the general validity of this body of research. We begin with a brief recap of the ontology-based framework, and the reasoning that went into its development. Essentially, the following section is a summary of the arguments developed in support of the hypotheses.

6.1.1 Summary: What was done and why?

At the outset of this thesis, we stated that the goal of this research is to address challenges in the realm of federated simulations. Distributed and federated simulations are developed to study the emergent behavior of large systems for which sub-system level models exist. While there are several aspects to achieving interoperability between the simulation models of different sub-systems, our focus is on attaining representational compatibility. In order to exchange information between simulations at run-time, a common information model defining the representation of all shared simulation concepts must be defined. Since federate simulations often employ disparate representations of

shared concepts, attaining representational compatibility in the entire federation tends to involve significant effort. Essentially, this entails either modifying each federate simulation model, or defining an interface that translates federate representations of concepts to a chosen common representation.

The goal of this research is to support the attainment of representational compatibility in a federated simulation, in a partially automated, reusable fashion. To meet this goal, we use ontologies to capture and reuse knowledge about simulation models. Essentially, we have shown that concepts in a given simulation model can be described in a formal, unambiguous fashion using an ontology. Since these simulation model descriptions are machine-interpretable, we have been able to design and employ procedures that use them to perform tasks involved with attaining representational compatibility in a federation. Our approach involves defining a federation-specific common information model based on the available descriptions of federate simulation models. Furthermore, we define procedural relationships between the federate and common descriptions of simulation model entities. By doing so, we make federate simulation models and their descriptions readily reusable. Defining relationships between federate and common simulation entities means that federate simulation models can represent entities in disparate ways and still exchange information with each other. Therefore, the same model can be used in several federations without having to modify its implementation. Also, their corresponding descriptions (ontologies) can be reused as input to the procedures that automate the process of arriving at a common information model.

To ensure that this approach is developed correctly and affirm its validity in solving the representation inconsistency problem, we have taken a structured, four-step approach that has been elaborated upon in previous chapters. We began by conducting a survey of related literature that ties into the research questions and hypotheses we posed. This survey was instrumental in helping us identify what the highlights and limitations of existing frameworks are in contrast with what we have proposed. Based on this, we identified ideas and implementations that can potentially be leveraged to support our cause. Next, we use the key points from the work of others as a basis upon which we built our framework. Analogous to having a strong foundation for a well-constructed edifice, building upon existing frameworks (such as HLA, schema mapping and morphisms) and theory (i.e. graph theory) builds confidence in the validity of our research developments. Having developed our framework, we demonstrated its usefulness in the context of an example federation development problem. By selecting this problem as one that is representative of the types of problems that the framework is meant to address, we are able to speak of the usefulness of our framework in supporting reuse and automation in developing federated simulations. Accepting the generality of this work involves taking a leap of faith to accept the general validity of this framework based on its empirical performance.

The framework that we have developed consists of several components that work together to achieve the overall goal of attaining representational compatibility in a federation.

- At a high level of abstraction, a metamodel for capturing simulation models in ontologies (the world ontology) is defined. This metamodel serves as a communal vocabulary, in terms of which all simulation concepts are defined. Therefore, the semantics of any given simulation concept, and hence the relationship between any two concepts can be unambiguously inferred. By basing the definition of this metamodel on the HLA OMT, we have leveraged the research conducted by others to determine how best to capture different simulation model concepts.

- Furthermore, we have defined a relationship class to capture relationships between two simulation concepts in an ontology. This class has been defined so as to include knowledge of both matches and mappings between simulation entities (recall, a match specifies *which* entities relate, a mapping specifies *how* they relate), as both are required to perform run-time information exchange.

- Finally, an algorithm to generate a common information model and procedures to translate information between federate and common representations is employed in this framework. This algorithm uses graph theory and associated algorithms as a means to automate the above-mentioned tasks in an efficient manner.

When compared to the traditional approach taken on in federation development, our framework offers clear advantages. Conventional methods of integrating federate simulations (such as the HLA FEDEP model) require federation developers to manually define a common information model, and modify the representations of individual

simulations so as to be consistent with this common information model. On the other hand, the ontology-based approach facilitates the selection of an appropriate common information model in an automated fashion. Clearly, this process is not completely automated—knowledge needs to be provided by the federation developer in the creation of SONTs and the specification of relationships between them, which entails a significant effort. Still, partial automation in support of achieving interoperability reduces the time and effort required to integrate a set of federate simulations.

The simplified reuse of federates and federations in the ontology-based framework helps to further facilitate federation development. Once an ontology corresponding to a given simulation model is created, the semantics contained within that ontology can be reused to integrate its corresponding simulation model into multiple federations. Therefore, the cost of developing a simulation ontology is amortized over its application in several FONT development problems. The point at which the cost of developing a SONT is recovered and the benefit of reuse is achieved depends on the reuse scenario. Recall that when a SONT is created, only the entities shared in a federation are modeled in that SONT. Based on the set of shared concepts in a given federation, an existing SONT corresponding to a federate simulation may need to be extended to include concepts that were previously not modeled. However, if an existing SONT already captures all shared entities in a federation, that SONT can be applied as-is. In such a case, the benefit of the ontology-based approach is reaped in reusing an existing SONT once.

Looking back at the overall goal of this research, we have accomplished what we set out to do. The framework that has been developed over the previous chapters supports automation and reuse in federation development, specifically in the process of attaining representational compatibility between interoperating simulations. Along the way, we have built-up arguments to support the claim that this work is valid; both in its structure and its performance. However, there are fundamental limitations that we have uncovered in doing so, which need to be recapitulated before we stake a claim as to the general validity of this research.

6.1.2 Limitations

There are fundamental limitations to the functionality of this framework that need to be explicitly stated. The framework we have developed *supports* automation and re-use in federation development; it does not completely automate this process, nor does it render federate simulations (or existing federations) readily reusable in a plug-and-play manner. There are several tasks involved in achieving interoperability in a simulation federation, all of which are not addressed in this thesis. For example, an interface between each simulator and an RTI needs to be developed, where services such as publishing and subscribing to entities in the federation are defined. This aspect of interoperability is not addressed in this research; we focus solely on developing a common information model and a set of transformation stubs so that information between federates can be exchanged in a consistent manner.

The degree to which the overall process of achieving representational compatibility is automated in this framework is dependent on the set of related entities in a federation. First, the process of identifying matches between simulation entities is not automated; the GRIT algorithm only generates a mapping (transformations) for a sub-set of matched entities. As we have noted, the only transformations that can be generated automatically are those between two conceptually equivalent entities. In other words, if two related simulation entities refer to the same concept but have different representations, only then can their transformation be generated automatically. All other transformation procedures must be specified manually (e.g. e.g. `radius= diameter/2` and `diameter= radius*2)`. In most cases, entities of two or more simulations that are coupled are different representations of the same concept. As the number of related, disparate concepts in a federation grows, so too does the effort required to attain representational compatibility, given that each of these transformations needs to be specified manually. Moreover, since transformations are specified in pairs, equivalent knowledge as to how two entities relate needs to be explicitly specified twice, which somewhat negates the extent to which knowledge is 'reused' in this framework.

Our implementation also imposes limitations on the extent to which automation is achieved. For example, we discovered in Chapter 5 that for any relationship between a primitive and custom data type, the associated transformations must be specified manually, since primitive data types have been defined such that they do not have any attributes. Similarly, transformations for related attributes with enumerated data types must also be specified explicitly. Furthermore, it is assumed that when a match between

two attributes is specified, a corresponding match between their data types is specified as well. The existence of data type relationships is not automated, and no error checking exists to make sure all required data type transformations have been specified. Theoretically, the resultant FONT could be incomplete and federation developers would have no knowledge of this until run-time, when a call to a non-existent data type transformation is made.

There are limitations concerned with the reusability aspect of this framework as well. We have made the claim that by defining transformation stubs to convert between federate and common representations of shared simulation entities, the same federate, and hence its corresponding SONT can be reused in multiple federations. In theory, a SONT describes a federate simulation model, and once defined, it can be reused every time the corresponding federate participates in a new federation. However, recall that in a SONT, we only model those concepts of a federate simulation that are expected to be shared with other federates. Realistically, it is not possible to know exactly which concepts in a federate simulation maybe shared in any given federation a-priori. On the other hand, it is wasteful to model every concept in a given simulation domain in its SONT. Therefore, when a simulation participates in multiple federations, its SONT may not be readily reusable; it may need to be modified or augmented. Still, an existing SONT provides a basis for such modification, and federation developers do not need to start from scratch.

The statements above apply to the reuse of federations. Existing federations may be extended by adding new federates and defining relationships between their shared entities

and those of others (or simply the corresponding common representation). Depending on the set of relationships involving this new federate, other federate SONTs may have to be modified to include a larger set of concepts. This occurs when there is a new aspect to the coupling and interplay between the existing and new federates that has not been modeled in the existing federation. Furthermore, existing selections of the common representation and associated transformations may need to be revised as well. Again, while this means that the existing federation is not readily reusable, they serve as a 'partially-complete' starting point for creating new federations.

Another fundamental limitation of this framework deals with its interaction with an RTI. In this framework, we support the achievement of representational compatibility by capturing a common information model and a set of transformation stubs to help facilitate consistent information transfer. The run-time exchange of information between federates in a consistent manner is ultimately dependent upon the application of the knowledge captured in a FONT by an RTI. That is, the RTI used to facilitate run-time information exchange in a federation developed using this framework must access the FONT and invoke transformation stubs in an appropriate manner. First, the RTI must mimic the information structure (objects and events) defined in the FONT as abstract data types in the selected OOP that the transformation stubs have been written in. The existing transformations stubs must then be compiled as procedures that pass instances of these abstract data types as input and output parameters. Finally, the RTI must be able to interpret which transformation procedure to invoke when a given federate entity is

published or subscribed. To do so, the RTI should be able to access and construe relationships between SONT and Common entities captured in the FONT.

It is important to note that no existing federated or distributed simulation RTI has been designed with capabilities to parse an ontology corresponding to a given federation. Most RTI's, such as that of the HLA, only transfer information published from a source federate and reflect that information in all subscribing target federates. There is no notion of converting between representations here; it is assumed that all federate simulations have been modified so as to represent entities in a communal form. However, research efforts are currently being undertaken in the development of a next-generation RTI, which complements the research presented in this thesis. An Extensible Framework for Interoperable Distributed Simulation (IDSim) is being designed so as to interface with a FONT and thereby facilitate the transformation of information from one form to another as it is exchanged during a federation's execution (Fitzgibbons and Fujimoto 2004). The interface between the ontology-based framework and IDSim is defined in terms of an XML information model that is equivalent to a FONT.

Finally, the object-oriented approach to modeling concepts in an ontology confines the limits to which this framework is applicable. The knowledge model based on which the World Ontology, SONTs and FONTs are defined bears close resemblance to the object oriented paradigm. Therefore, all simulation entities modeled in this framework are done so in an object oriented fashion. That is, Objects, Events and Data types, collectively classes, are all defined in terms of a set of member slots, namely simulation attributes.

This means that every federate simulation model must captured in its corresponding ontology in an object oriented fashion, irrespective of whether the underlying simulation model employs such a representation. For legacy simulations written in languages such as Fortran (Nyhoff and Sanford 1995), object-oriented wrappers must be build around them, based on which their corresponding SONTs are modeled. Developing these object-oriented wrappers adds to the time and effort required to achieve interoperability in a federation, and hence works against our overall goal of reducing the cost of achieving interoperability.

Having identified the highlights and fundamental limitations of this framework we may now address the validity of this body of research and the hypotheses put forth in Chapter 1. In previous sections, we have performed validation of the individual parts of this research from the standpoint of their structure and performance (in the context of a single example), as illustrated in the Validation Square (Figure 1.4) (Pedersen, Emblemsvag, Bailey et al. 2000). In the following section, we bring all previous arguments together and present a case for accepting the general validity of this framework as a whole. The limitations discussed above help to define bounds on the validity of our hypotheses.

6.1.3 Theoretical Performance Validity

In Chapter 1, an overall research question was posed as to how the process of achieving representation compatibility in a federation could be automated (research question 1). This question was broken down into two constituent sub-questions (research questions 2 and 3). Their corresponding hypotheses address individual portions that together

comprise hypothesis 1 (they deal with the capture of knowledge and its subsequent application to support automation, respectively). In this discussion about the validity of this body of research, we begin by consolidating arguments backing the validity of hypotheses 2 and 3. Having accepted the validity of these hypotheses, we then build upon their supporting arguments so as to present a case for the validity of the overall hypothesis proposed in connection with research question 1.

Question 2: How should simulation concepts be represented in an ontology to support achieving interoperability?

Hypothesis 2: A metamodel for specifying simulation ontologies can be developed. The set of concepts and relationships between them defined in this metamodel form a vocabulary for describing simulation ontologies. If all simulation concepts are modeled using the same vocabulary, the relationships between two coupled simulation concepts in a federation can be inferred in an automated fashion.

In hypothesis 2, we proposed that a metamodel could be developed as a common vocabulary for describing simulation models in ontologies. If all federate simulations were described in terms of the same common vocabulary, the relationships between them could be inferred automatically. To validate this claim, we have developed this hypothesis into the World Ontology, and tested its expressiveness in the context of the Air Traffic federated simulation. The World Ontology specification stemmed from the HLA OMT, which is a well-developed template for specifying simulation and federation object models (SOMs and FOMs) within the HLA framework. This template has been developed taking multiple simulation coupling scenarios into consideration, is currently

part of an IEEE specification (IEEE 2000), and is used to support federation development within the DoD HLA environment. By leveraging the HLA OMT, we have given a sound, thoroughly investigated and valid basis to the World Ontology. Further, we have successfully tested the performance of this metamodel in capturing three federate simulation models, namely the ATC, GTC and Ground Services federates. Since we have determined that the shared entities of these simulations are representative of the different types of entities that the World Ontology is meant to be able to model (such as objects, events and custom data types), we project that the World Ontology is able to capture all federate simulation models. To accept the usefulness of this metamodel outside the context of the air traffic simulation, we have to take a 'leap of faith'—while it is not viable to test its usefulness on a large range of federation development scenarios, we are fairly confident that the successful modeling of the air traffic federation will be reflected in most other cases.

Of course, the usefulness and applicability of the World Ontology is limited. We make the assumption that all simulation models employ an object oriented representation. Furthermore, we stake the claim that relationships between federate simulations can be inferred based on the fact that they are described in terms of the same vocabulary. To do so, additional knowledge is required to be specified, such as the existence of matches between objects, events and data types. The mapping between two or more entities can only be correctly inferred if two matched entities are conceptually equivalent, all others must be explicitly specified. Given that all additional knowledge required is specified (and the fact that this framework allows for the specification of this additional

knowledge), we accept the general validity of hypothesis 2. The key points in support of this are:

- ✓ The World Ontology specification is leveraged from the HLA OMT, a well-accepted template for defining simulation information models
- ✓ The World Ontology has successfully been used to capture shared concepts of the ATC, GTC and Ground Services simulation models
- ✓ Relationships between simulation entities can be (and have been) successfully inferred when they are specified in terms of World Ontology constructs.

The general validity of this hypothesis is bounded by the following characteristics:

- ✓ An object oriented representation of a federate simulation model exists
- ✓ Additional knowledge required to infer relationships (mappings) between simulation entities is explicitly provided.

Question 3: How can the transformations between two representations of a simulation concept be derived in an automated fashion?

Hypothesis 3: Relationships between federate simulation entities are captured in terms of a relationship with their common, federation-level representation. The relationships between concepts defined in the simulation ontology metamodel can be composed together to derive the federate-common entity relationships. An algorithm can be developed to generate a connected graph of existing relationships in the federation. Graph traversal algorithms can be developed to identify relationships between simulation entities a chain of these existing relationships.

The specification of a common representation for all shared federate simulation entities and the generation of federate—common transformations are proposed in hypothesis 3. Here, we state that by using a graph traversal approach, we can select a common representation from amongst the set of related federate entities, and subsequently map federate entities to and from their common representations in an automated fashion. In Chapter 4, we have presented GRIT, a graph-based algorithm to accomplish exactly these tasks. The instantiation of a common representation and associated transformations in this algorithm is based on the knowledge of the shortest path between two nodes in a directed graph. We have employed Dijkstra's graph algorithm, an efficient, well-accepted algorithm to identify the shortest path between nodes in the graphs corresponding to a FONT (Dijkstra 1959). The usefulness of this algorithm to automate the specification of a FONT has been tested in the context of the air traffic federation. Here, we traversed through different mapping scenarios to study the correctness and efficiency of this algorithm. It was shown that following the steps in the GRIT algorithm, a common representation for shared simulation entities is selected that leads to the simplest

composition of the resultant federate—common relationships, wherein the loss of information in these transformations is minimal. Having demonstrated the GRIT algorithm's use in automatically deriving a suitable common information model and correctly generating the subsequent federate—common transformation stubs in the context of a significantly complex example, we project that it will perform in a similar fashion when used in the development of other federation ontologies.

Again, there are inherent limitations and assumptions associated with the performance of this algorithm. In order for the correct common representation to be selected, knowledge as to the lossiness of different transformations must be specified by the federation developer. Without this knowledge, one cannot assess the differences in lossiness, and therefore one cannot determine which representation is best chosen as common. Moreover, we make the assumption that for every collection of matched entities, a match between their value-types exists as well. Perhaps the biggest limitation of this algorithm is that it only generates mappings for relationships between conceptually equal simulation entities. All relationships between simulation entities involving anything more than a representation conversion must be explicitly specified. Under the assumption that a minority relationships in federated simulations are between disparate concepts, the GRIT algorithm is clearly useful in reducing the time and effort required to attain representational compatibility between a set of federate simulations. Hence, we accept hypothesis 3 to be valid, based on the following key arguments:

- ✓ The algorithm to automatically generate transformations between related simulation entities is founded in the theory of graphs and Dijkstra's widely-accepted graph algorithm

- ✓ In the context of a significantly complicated federation development problem, we have shown that the GRIT algorithm can automatically select a suitable common information model and compose associated transformations using the knowledge contained in a federation ontology.

The characteristics that bound the validity of hypothesis 3 are:

- ✓ The selection of a common information model requires a federation developer(s) to specify knowledge as to the loss of information in transformations for every (attribute-level) match specified

- ✓ Only a subset of the required transformations can be generated automatically. All transformations corresponding to relationships between disparate concepts must be explicitly defined.

- ✓ We have assumed that the best selection of a common representation is one that leads to the fewest end to end lossy transformations, without taking into account the extent of information loss (two slightly lossy transformations may be better than one very lossy transformation).

Question 1: How and to what extent can the process of achieving representational compatibility between simulations in a federation be automated?

Hypothesis 1: Ontologies can be used to formally describe the semantics of concepts in a federate simulation model. These semantics can then be applied to generate a required common information model and associated transformation stubs in a partially automated fashion.

Based on the validity of hypotheses 2 and 3, we can accept the overall hypothesis posed in this thesis to be valid. In hypothesis 1, our vision for an ontology-based framework to support achieving interoperability between federate simulations was presented. Through the length of this thesis, we have developed this vision and tested its performance in the context of a quintessential example. Specifically, we have shown that (i) ontologies can be used to describe the semantics of a simulation domain (ii) the procedural relationships between federate simulation entities can be generated and captured in an automated fashion. The validity of these functional components has been discussed, characterized and accepted in terms of the first three quadrants of the Validation Square. Essentially, the validity of hypotheses 2 and 3 together implies that our overall vision is valid. *The framework we have developed can be used to support the development of simulation federations, in general, in a more cost and time effective manner. Using a communal vocabulary, shared concepts of federate simulation domains are modeled in ontologies in an object-oriented fashion. With the help of a graph-theoretic algorithm, a common, federation-level information model for shared entities and associated transformation stubs can be generated automatically.* As we have previously discussed, the usefulness of this framework is dependent on the existence of an RTI that applies the knowledge

contained in a federation ontology to facilitate consistent information transfer in a consistent manner.

6.2 Future work

It is said that every answer leads to new questions—this is certainly true in the case of this research. In answering the research questions and validating our hypotheses, we have identified certain limitations to our research. Exploring ways to combat these limitations entails posing and investigating additional research questions. In this section, we present a brief overview of the paths along which we see this work progressing. It is important to highlight these, as we would like to extend the confines to which the usefulness of this body of research is limited.

6.2.1 Extending the limits of reusability and automation

The overall goal of this investigation has been to reduce the cost and time required to achieve interoperability between simulations. While we have been able to automate part of this process, there is still room for automation to a greater degree. The same is true with regards to the extent to which we support reuse in federation development. One avenue for further automation is in the specification of matches between entities of federate simulations. In Chapter 2, we evaluated several frameworks and algorithms that have been designed to find matching entities across disparate schemas. For example, PROMPT (Noy and Musen 2000) is an algorithm developed to identify classes and slots of two ontologies that potentially relate to each other. Similarly, GLUE (Alagic and Bernstein 2001) uses machine learning techniques to identify strong relationships

between disparate information schemas. These systems could be employed to identify matches between the entities modeled in the different SONTs corresponding to federate simulations that need to interact with each other. Automated entity matching would mean that users of the ontology-based framework would no longer explicitly have to specify relationships between SONT entities; they would simply approve or decline automatically generated 'suggested' matches. The knowledge of existing, approved matches could then be used to refine the set of subsequent suggested matches. Clearly, this reduces both time and effort required to integrate federate simulations, and moves us one step closer to plug-and play reuse of existing federates (and their SONTs) in multiple federations.

Another avenue for future development of the ontology-based framework is in the specification of relationships between disparate simulation concepts. Currently, the transformations for all such relationships need to be specified manually. To do so, the same knowledge (a relationship between two entities) must be specified in two transformation procedures, which is wasteful. Perhaps the knowledge as to the conceptual relationship between two simulation entities could be provided only once, as a non-causal, declarative equation. From this relationship, the required causal procedures could be generated automatically using declarative equation solvers such as Maple. Such an approach to relationship specification would eradicate the need for federation developers to specify transformations explicitly. Specifying an equation describing the relationship between two or more SONT entities involves less effort and repetition, and hence would contribute towards a greater extent of automation and reuse in this framework. Clearly,

implementing the specification of declarative relationships implies that a richer set of semantics be defined to describe relationships (compared to the current specification of the relationship class in the World Ontology). Determining what this set of semantics should be is an interesting research question in itself.

6.2.2 Going beyond the confines of Federated Simulation

Although the framework developed in this thesis is rooted in the domain of distributed simulation, the ideas developed here could potentially be applied to address a much larger challenge—that of system (product) information management. As the market place for engineered products, processes and services becomes more competitive, the need to capture, store and reuse information and knowledge related to the design and development of these systems has increased. Any system development team capable of efficiently managing and reusing their information and knowledge capital holds and edge over competitors, and is able to develop better end-products and bring them to market faster. Therefore, the management of information and knowledge has become an important challenge in the field of Product Lifecycle Management (PLM). Traditionally, information about various aspects of a system's lifecycle is maintained in different electronic repositories (Figure 6.1). However, there is significant overlap and coupling between the different aspects of a system's design, development, use and dissolution. Hence, relationships exist between the different knowledge and information-bases associated with myriad aspects of a system's lifecycle. These information sources are all managed by different owners and may employ distinct representations of system-related concepts. Therefore, a significant issue in system information management is reconciling

between different representations of information about equivalent or related concepts in multiple repositories.

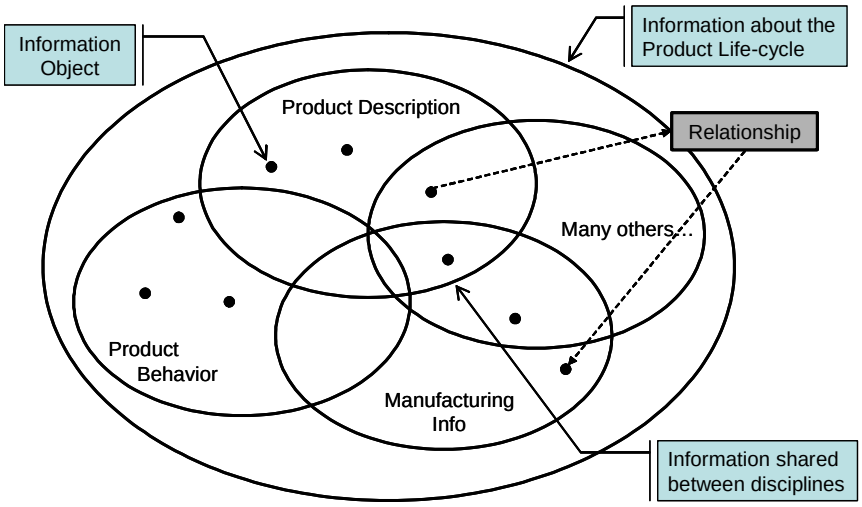


Figure 6.1: Shared Information between Various Aspects of a Product's Lifecycle

In this body of research, we have essentially presented a framework in which different information models can be related to each other. Clearly, the ideas we have developed are applicable to help address the issue of relating between different product-related information sources. By generating transformations to convey information across different, overlapping repositories, the burden of managing system lifecycle related information across disparate sources (schemas) can be significantly lightened. The framework we have developed could be used as a starting point to realize a system that serves this purpose. Applying the ontology-based framework to the significantly larger scope of system information management requires significant modification and extension. Therefore, answering the question *'How can a framework for capturing and*

relating simulation model concepts using ontologies be extended to support the management of product lifecycle related information' could entail a meaningful investigation and result in a useful research contribution. In this context, perhaps the most important of issues to be investigated is the development of a vocabulary (World Ontology) that is expressive enough that it may be used to describe the entire range of concepts associated with a product's lifecycle.

6.3 Closing Statements

There is immense potential in the use of semantic technologies to capture human knowledge and apply it to automate tasks involved in the design and development of complex systems. In this thesis, we have taken the first steps in exploiting this potential to facilitate automation in the development of federated, system level simulations. In doing so, we have discovered that this work has fundamental limitations, and that there are several avenues along which it can still be extended. As the domain of knowledge-based systems and semantically-rich information modeling moves past its inception, we aspire that the steps we have taken may be furthered such that the usefulness of this contribution may proliferate to a wider audience.

REFERENCES

Aughenbaugh, J., Paredis, C. (2004). "The Role and Limitations of Modeling and Simulation in Systems Design." *2004 ASME International Mechanical Engineering Congress and R&D Expo: Computers & Information in Engineering*, Anahiem, CA.

Base Object Model Study Group (2001). "BOM Methodology Strawman (BMS) Specification.", Simulation Interoperability Standards Organization, Orlando, FL.

Berners-Lee, T., J. Hendler and O. Lassila (2001). "The Semantic Web." *The Scientific American Magazine* (279)

Bernstein, P. (2003). "Applying Model Management to Classical Meta Data Problems." *in the Proceedings of Conference on Innovative Data Systems Research (CIDR-2003)*, 209-220.

Bondy, J. and U. Murthy (1981). *Graph Theory with Applications*, North-Holland.

Cellier, F. E. (1991). *Continuous System Modeling*. New York, NY, Springer-Verlag.

Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P.D., Rice, J.P. (1998). "OKBC: A Programmatic Foundation for Knowledge Base Interoperability." *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, AAAI Press.

Chen, P. (1976). "The Entity Relationship Model - Towards a Unified View of Data." *ACM Transactions on Database Systems*, **1**: 9-36.

Dahmann, J., Fujimoto, R., Weatherly, R. (1997). "The Department of Defense High Level Architecture." *1997 Winter Simulation Conference*, 142-149.

Dahmann, J., Salisbury, M., Turrell, C., Barry, P., Blemberg, P. (1999). "HLA and Beyond: Interoperability Challenges." *Fall 1999 Simulation Interoperability Workshop*, Fall, 1999.

Defense Modeling and Simulation Office (DMSO) (1999). "High Level Architecture: Federation Development and Execution Process (FEDEP) Model."

Defense Modeling and Simulation Office (DMSO) (2004). "High Level Architecture Website" (www.dsmo.mil/public/transition/HLA).

Dijkstra, E. W. (1959). "A Note on Two Problems in Connection with Graphs." *Numerische Mathematics*, **1**: 269-271.

Doan, A., Domingos, P. and Halvey, A. (2001). "Reconciling Schemas of Disparate Data Sources: A Machine Learning Approach." *Proceedings of the ACM SIGMOD Conference*, 509-520.

Even, S. (1979). *Graph Algorithms*. Rockville, MD, Computer Science Press, Inc.

Fitzgibbons, J. and R. Fujimoto (2004). "IDSim: An Extensible Framework for Interoperable Distributed Simulation." *In the proceedings of the International Conference on Web Services*, San Diego, CA, July, 2004.

Floyd, R. W. (1962). "Algorithm 97: Shortest Path." *Communications of the ACM*, **5**: 345.

Forsberg, K. and H. Mooz (1992). "The Relationship of Systems Engineering to the Project Cycle." *Engineering Management Journal*, **4**(3): 36-38.

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

Genesereth, M. (1995). "Knowledge Interchange Format Specification.", Stanford University. **(**http://logic.stanford.edu/kif/).

Gruber, T. (1993). "Toward Principles for the Design of Ontologies Used for Knowledge Sharing." *International Workshop on Formal Ontology*, Padova, Italy, March, 1993.

Horvath, I. and Van Der Vegte, W. (2003). "Nucleus-based Product Conceptualization - Part 1: Principles and Formalization." *International Conference on Engineering Design (ICED 03)*, Stockholm, Sweden, August 19-21, 2003.

IEEE (2000). "Std 1516.1-2000, Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification."

IEEE (2000). "Std 1516.2-2000, Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Object Model Template (OMT) Specification."

IEEE (2000). "Std 1516.3-2000, Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules."

Kasyanov, V. and Evstigneev, V. (1994). *Graph Theory for Programmers: Algorithms for Processing Trees*. Dordrecht, The Netherlands, Kluwer Academic Publishers.

Kuhl, F., Dahmann, J. and Weatherly, R. (1999). *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Upper Saddle River, NJ, Prentice Hall.

Kuperberg, G. (2000). "Dept. of Mathematics Glossary.", University of California. (http://www.math.ucdavis.edu/profiles/glossary.html).

Lassila, O. and McGuinness, D. (2001). "The Role of Frame-Based Representation on the Semantic Web.", Knowledge System Laboratory, Stanford University, Stanford, CA.

Li, W. and Clifton, C. (2000). "SEMINT: A Tool for Identifying Attribute Correspondences in Heterogeneous Databases using Neural Networks." *Data and Knowledge Engineering*, **33**(1): 49-84.

Liang, V. and Paredis, C. (2004). "A Port Ontology for Conceptual Design of Systems." *Journal of Computing and Information Science in Engineering*, **4**(3): tba.

Lutz, B. (1999). "FEDEP V1.4: An Update to the HLA Process Model." *Winter Simulation Conference*.

Macannuco, D., B. Dufault and L. Ingraham (1998). "An Agile FOM Framework." *Simulation Interoperability Workshop*, September, 1998.

Macannuco, D., Coffin, D., Dufault, B. and Civinskas, W. (1999). "Experiences with a FOM Agile Federate." *Simulation Interoperability Workshop*, March 14-19, 1999.

Madhavan, J., Bernstein, P., Domingos, P. and Halevy, A. (2002). "Representing and Reasoning about Mappings between Domain Models." *American Association for Artificial Intelligence (AAAI)*.

Madhavan, J., Bernstein, P. and Rahm, E. (2001). "Generic Schema Management with Cupid." *Proceedings of the Very Large Data Base Conference*, Rome, Italy.

Maedche, A., Motik, B. and Stojanovic, L. (2003). "Managing Multiple and Distributed Ontologies on the Semantic Web." *International Journal on Very Large Data Bases*, **12**: 286-302.

Malone, B., Papay, M. (1999). "ModelCenter: An Integration Environment for Simulation Based Design." *1999 Fall Simulation Interoperability Workshop*.

Miller, G. and Filipelli, L. (1999). "An XML Representation of HLA Object Models." *Simulation Interoperability Workshop*.

Milo, T. and Zohar, S. (1998). "Using Schema Matching to Simplify Heterogeneous Data Translation" *Proceedings of the International Conference on Very Large Data Bases*: 122-133.

Minsky, M. (1975). *A Framework for Representing Knowledge*, McGraw-Hill.

Mitra, P., Wiederhold, G. and Jannink, J. (1999). "Semi-Automatic Integration of Knowledge Sources." *Proceedings of FUSION 99*, Sunnyvale, CA.

Morse, K. (1996). "Interest Management in Large Scale Distributed Simulations.", University of California, Irvine, CA.

Naiburg, E. and R. Maksimchuk (2001). *UML for Database Design*. Upper Saddle River, NJ, Addison-Wesley.

Novak, G. S. (1995). "Conversion of Units of Measurement." *IEEE Transactions on Software Engineering*, **21**(8): 651-661.

Noy, N., Fergerson, R. and Musen, M. (2000). "The Knowledge Model of Protégé 2000: Combining Interoperability and Flexibility.", Stanford Medical Informatics Technical Report, Stanford University, Stanford, CA.

Noy, N. and M. Musen (2000). "PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment." *Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, Austin, TX.

Noy, N. and Musen, M. (1999). "An Algorithm for Merging and Aligning Ontologies: Automation and Tool Support." *Sixteenth National Conference on Artificial Intelligence (AAAI99)*, Orlando, FL.

Nyhoff, L. and L. Sanford (1995). *FORTRAN 77 for Engineers and Scientists*, Pearson Education.

Pahl, G. and W. Beitz (1996). *Engineering Design - A Systematic Approach*. London, UK, Springer Verlag London Ltd.

Peak, R. S. (2003). "Characterizing Fine-Grained Associativity Gaps: A Preliminary Study of CAD-CAE Model Interoperability."*ASME Design Engineering Technical Conference and Computers and Information in Engineering Conference*, Chicago, IL, September, 2003, ASME.

Pedersen, K., Emblemsvag, J., Bailey, R., Allen, J. and Mistree, F. (2000). "Validating Design Methods and Research: The Validation Square." *Proceedings of the 2000 ASME Design Engineering Technical Conferences*, Baltimore, MD, September 10-14, 2000.

Rahm, E. and Bernstein, P. (2001). "On Matching Schemas Automatically." *International Journal on Very Large Data Bases*, **10**(4).

Ryan, P. J. (1992). "An Approach to Asymptotic Complexity." *Mathematics and Computer Education*, **26**: 135-46.

Ryde, M. and Taylor, S. (2003). "Issues Using COTS Simulation Software Packages for the Interoperation of Models." *Proceedings of the 2003 Winter Simulation Conference*.

Scrubber, R., Lutz, R. and Dahmann, J. (1998). "Automation of the Federation Development and Execution Process." *1998 Fall Simulation Interoperability Workshop*, Fall, 1998.

Seepersad, C. (2001). *The Utility-Based Compromise Decision Support Problem with Applications in Product Platform Design*. Thesis. The G.W. Woodruff School of Mechanical Engineering, Georgia Institute of Technology. Atlanta, GA.

Szykman, S., R. D. Sriram and W. C. Regli (2001). "The Role of Knowledge in Next-generation Product Development Systems." *ASME Journal of Computing and Information Science in Engineering*, **1**.

TopQuadrant (2003). "Semantic Integration: Strategies and Tools.", TopQuadrant Inc., Beaver Falls, PA.

Turrell, R., Bouwens, C. and McCormack, J. (1999). "HLA Federation Development and Execution: Automated End-to-End Support of the FEDEP with the HLA Tool Suite." *1999 Spring Simulation Interoperability Workshop*, Spring, 1999.

Walmsley, P. (2001). *Definitive XML Schema*. Upper Saddle River, NJ, Prentice Hall.

World Wide Web Consortium (W3C) (2003). "XQuery 1.0: An XML Query Language Working Draft."(http://www.w3.org/TR/xquery/).

World Wide Web Consortium (W3C) (2004). "RDF Primer" (http://www.w3.org/TR/rdf-primer/).

Yellen, J. and G. Gross (1998). *Graph Theory and its Applications*, CRC Press.
Zeigler, B. (1990). *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*. Boston, MA, Academic Press.