UNIVERSITÉ DE MONTRÉAL

AUTOMATING SYSTEM-LEVEL DATA-INTERCHANGE SOFTWARE THROUGH A SYSTEM INTERFACE DESCRIPTION LANGUAGE

MARTIN TAPP

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION

DU DIPLÔME DE PHILOSOPHIÆ DOCTOR

(GÉNIE INFORMATIQUE)

DÉCEMBRE 2013

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

AUTOMATING SYSTEM-LEVEL DATA-INTERCHANGE SOFTWARE THROUGH A SYSTEM INTERFACE DESCRIPTION LANGUAGE

présentée par: TAPP Martin

en vue de l'obtention du diplôme de : Philosophiæ Doctor

a été dûment acceptée par le jury d'examen constitué de :

Mme BOUCHENEB Hanifa, Doctorat, présidente
Mme NICOLESCU Gabriela, Doct., membre et directrice de recherche
M. ABOULHAMID El Mostapha, Ph.D., membre et codirecteur de recherche
M. DAGENAIS Michel, Ph.D., membre
Mme MORSE Katherine L., Ph.D., membre

DEDICATION

To the ones following an idea through.

ACKNOWLEDGMENTS

I wish to thank CAE for supporting me in undertaking this endeavor and making it possible. I particular want to acknowledge the following persons for supporting me: Éric Simon, Louis Dontigny, Éric Bouthillier, Charles Fortier, Sébastien Lévesque, and particularly Jean-François Campeau for seeing the value in this work, and supporting the required experimentation to demonstrate it. I want to acknowledge my fellow architects and co-workers, notably Alexandre, Marc-André, Steve, Benoit, and Yannick, for their precious support and feedback, and above all Sidney Chartrand for his many insights and contributions to this work which at the same time inspired me.

I wish to thank Gabriela Nicolescu and El Mostapha Aboulhamid for their support and guidance along with giving me the freedom to pursue this work. Their valuable advice contributed to making me a better researcher.

I wish to thank all of the members of the revision committee for the time they put into revising this work.

I would also like to acknowledge the SISO community which many of its members helped shape this work.

Finally, to the most important persons in my life, my marvelous wife, Caroline, my wonderful daughters, Clara and Abigaël, my mom, Jacqueline, my dad, Gervais, my sister, Sophie, and my brothers, Julien and Rémi, along with Hélène, Marie-Claude, Julie, Ginette, and Yves, I wish to gracefully bow before you, and thank you for your continued support, inspiration, and unconditional love.

RÉSUMÉ

Les plates-formes d'aujourd'hui, telles que les simulateurs de missions (FMS), présentent un niveau sans précédent d'intégration de systèmes matériels et logiciels. Dans ce contexte, les intégrateurs de systèmes sont confrontés à une hétérogénéité d'interfaces système qui doivent être alignées et reliées ensemble afin de fournir les capacités prévues d'une plate-forme. Le seul aspect des échanges de données système est problématique allant de données désalignées jusqu'à des environnements multi-architecturaux utilisant différents types de protocoles de communication. Les intégrateurs sont également confrontés à des défis similaires lors de l'interaction de multiples plates-formes ensemble à travers des environnements de simulation distribuée où chaque plate-forme peut être considérée comme un système avec sa propre interface distincte. D'autre part, permettre la réutilisation de système à travers diverses plates-formes en support aux gammes de produits est un défi pour les fournisseurs de systèmes, car ils doivent adapter leurs interfaces système à des plates-formes hétérogènes faisant donc face aux mêmes difficultés que les intégrateurs. En outre, l'introduction de modifications aux interfaces système afin de répondre aux besoins tardifs d'affaires, ou à des contraintes de performance imprévues, par exemple, est d'autant plus ardue que leurs impacts sont difficiles à prévoir et que leurs effets sont souvent décelés tard dans le processus d'intégration.

En conséquence, cette thèse aborde la nécessité de simplifier l'intégration et l'interopérabilité système afin de réduire leurs coûts associés et d'accroître leur efficacité ainsi que leur efficience. Elle est destinée à apporter de nouvelles avancées dans les domaines de l'intégration système et de l'interopérabilité système. Notamment, en établissant une taxonomie commune, et en augmentant la compréhension des interfaces système, des divers aspects impactant les échanges de données système, des considérations des environnements multi-architecturaux, ainsi que des facteurs permettant la gouvernance d'interface ainsi que de la réutilisation système. À cette fin, deux objectifs de recherche ont été formulés.

Le premier objectif vise à définir un langage utilisé pour décrire les interfaces système et les divers aspects entourant leurs échanges de données. Par conséquent, trois aspects principaux sont étudiés relatifs aux interfaces système: les éléments de langage pertinents utilisés pour les décrire, la modélisation des interfaces système avec ce langage, et la capture des considérations multi-architecturales.

Le second objectif vise à définir une méthode pour automatiser le logiciel responsable des échanges de données système comme moyen pour simplifier les tâches impliquées dans l'intégration et l'interopérabilité système. Par conséquent, les compilateurs de modèles et les techniques de génération de code sont étudiés.

La démonstration de ces objectifs apporte de nouvelles avancées dans l'état de l'art de l'intégration système et de l'interopérabilité système. Notamment, ceci culmine en un nouveau langage de description d'interface système, SIDL, utilisé pour capturer les interfaces système et les divers aspects entourant leurs échanges de données, ainsi qu'en une nouvelle méthode pour automatiser le logiciel d'échange de données au niveau système à partir des interfaces systèmes capturées dans ce langage.

L'avènement de SIDL contribue également une nouvelle taxonomie fournissant une perspective complète sur l'interopérabilité système ainsi qu'en un langage commun qui peut être partagé entre les parties prenantes, tels que les intégrateurs, les fournisseurs et les experts système. Étant agnostique aux architectures, SIDL fournit un seul point de vue architectural supervisant toutes les interfaces système et capture les considérations multi-architecturales ce qui n'a jamais été réalisé avant ce travail. D'autant plus, un générateur de code SIDL est introduit présentant la nouveauté de générer le logiciel d'échange de données à partir d'un bassin plus riche d'information, notamment à partir des relations système de haut niveau allant jusqu'au bas niveau couvrant les détails protocolaires et d'encodage. En raison des considérations multi-architecturales qui sont capturées nativement dans SIDL, ceci permet au générateur de code d'être agnostique aux architectures le rendant réutilisable dans d'autres contextes.

Cette thèse ouvre également la voie à de futures recherches bâtissant sur ses contributions. Elle propose même une vision pour le développement d'applications logicielles avec comme objectif final de repousser encore plus loin les limites de la simplification et de l'automatisation des tâches liées à l'intégration et à l'interopérabilité système.

ABSTRACT

Today’s platforms, such as full mission simulators (FMSs), exhibit an unprecedented level of hardware and software system integration. In this context, system integrators face heterogeneous system interfaces which need to be aligned and interconnected together in order to deliver a platform's intended capabilities. The sole aspect of the data systems exchange is problematic ranging from data misalignment up to multi-architecture environments over varying kinds of communication protocols. Similar challenges are also faced by integrators when interoperating multiple platforms together through distributed simulation environments where each platform can be seen as a system with its own distinct interface. On the other hand, enabling system reuse across multiple platforms for product line support is challenging for system suppliers, as they need to adapt system interfaces to heterogeneous platforms therefore facing similar challenges as integrators. Furthermore, the introduction of system interface changes in order to respond to late business needs, or unforeseen performance constraints for instance, is even more arduous as impacts are challenging to predict and their effect are often found late into the integration process.

Consequently, this thesis tackles the need to simplify system integration and interoperability in order to reduce their associated costs and increase their effectiveness along with their efficiency. It is meant to bring new advances in the fields of system integration and system interoperability. Notably, by establishing a common taxonomy, and by increasing the understanding of system interfaces, the various aspects impacting system data exchanges, multi-architecture environment considerations, and the factors enabling interface governance as well as system reuse. To this end, two research objectives have been formulated.

The first objective aims at defining a language used to describe system interfaces and the various aspects surrounding their data exchanges. Therefore, three key aspects are studied relating to system interfaces: the relevant language elements used to describe them, modeling system interfaces with the language, and capturing multi-architecture considerations.

The second objective aims at defining a method to automate the software responsible for system data exchanges as a way of simplifying the tasks involved in system integration and interoperability. Therefore, model compilers and code generation techniques are studied.

The demonstration of these objectives brings new advances in the state of the art of system integration and system interoperability. Notably, this culminates in a novel system interface description language, SIDL, used to capture system interfaces and the various aspects surrounding their data exchanges, as well as a new method for automating the system-level data-interchange software from system interfaces captured in this language.

The advent of SIDL also contributes a new taxonomy providing a comprehensive perspective over system interoperability as well as a common language which can be shared amongst stakeholders, such as integrators, suppliers, and system experts. Being architecture-agnostic, SIDL provides a single architectural viewpoint overseeing all system interfaces and capturing multi-architecture considerations which was never achieved prior to this work. Furthermore, a SIDL code generator is introduced which has the novelty of generating the data-interchange software from a richer pool of information, notably from the high-level system relationships down to the low-level protocol and encoding details. Because multi-architecture considerations are captured natively in SIDL, this enables the code generator to be architecture-agnostic making it reusable in other contexts.

This thesis also paves the way for future research building upon its contributions. It even proposes a vision for software application development with the end goal being to push further the boundaries of simplifying and automating the tasks involved in system integration and interoperability.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AADL	Architecture Analysis & Design Language
ADL	Architecture Description Language
ANDEM	Architecture Neutral Data Exchange Model
AoS	Array of Structure
ASN.1	Abstract Syntax Notation One
BOM	Base Object Model
CDM	Conceptual Data Model
CGF	Computer Generated Forces
COTS	Commercial off-the-shelf
CTIA	Common Training Instrumentation Architecture
CPU	Central Processing Unit
DDS	Data Distribution Service
DEM	Data Exchange Model
DIS	Distributed Interactive Simulation
DMSO	Defense Modeling and Simulation Office
DoD	Department of Defense
DSEEP	Distributed Simulation Engineering and Execution Process
DSL	Domain-Specific Language
FACE	Future Airborne Capability Environment
FMS	Full Mission Simulator
FOM	Federation Object Model
GPU	Graphics Processing Unit

HLA	High-Level Architecture
IDL	Interface Definition Language
LCIM	Levels of Conceptual Interoperability Model
LDM	Logical Data Model
LVC	Live Virtual Constructive
OS	Operating System
OEM	Original Equipment Manufacturer
OMT	Object Model Template
OWL	Web Ontology Language
M&S	Modeling and Simulation
MEP	Message Exchange Pattern
QoS	Quality of Service
RCS	Radar Cross-Section
RDF	Resource Description Framework
RPR-FOM	Real-time Platform Reference Federation Object Model
RTI	Run-Time Infrastructure
SDEM	Simulation Data Exchange Model
SDM	Specific Data Model
SIDL	System Interface Description Language
SIMD	Single Instruction, Multiple Data
SIMNET	Simulator Network
SISO	Simulation Interoperability Standards Organization
SME	Subject Matter Expert
SoA	Structure of Array

SOA	Service-Oriented Architecture
SQL	Structured Query Language
SysML	Systems Modeling Language
TDL	TENA Definition Language
TENA	Test and Training Enabling Architecture
UoP	Unit of Portability
UML	Unified Modeling Language
USB	Universal Serial Bus
XMI	XML Metadata Interchange
XML	Extensible Markup Language
WSDL	Web Services Description Language

LIST OF APPENDICES

Part I

INTRODUCTION AND BACKGROUND

INTRODUCTION

Background - Better System Interoperability and Reuse

A full mission simulator (FMS), as illustrated in Figure 1, replicates an existing aircraft and its environment in order to provide training to aircraft crews. This requires the interaction of several hardware and software systems examples of which include: systems simulating aircraft flight and propulsion; systems replicating the environment surrounding the aircraft such as weather, motion, and air traffic; systems dealing with cockpit displays and pilot inputs; and systems supporting the training lessons providing instructor feedback and control. Moreover, these systems need to exchange data with each other, for instance, in order to replicate the end functions of the aircraft to the aircrew. In addition, multiple FMSs can be joined together to simulate air traffic or to perform joint missions involving multiple aircrews training together.

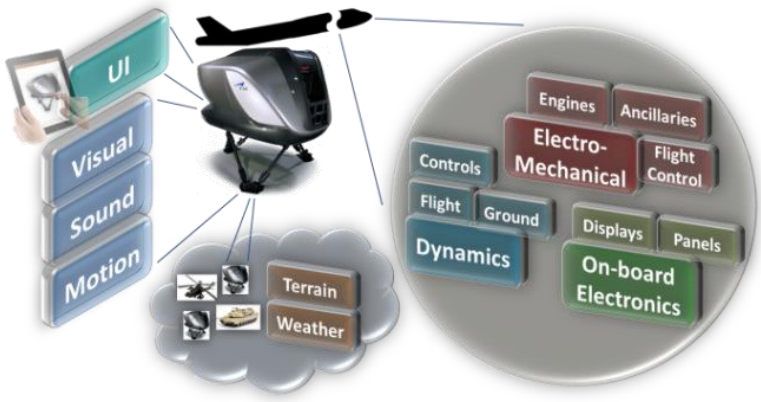

Figure 1: FMS Hardware and Software Systems

Today’s platforms, such as full mission simulators (FMSs), exhibit an unprecedented level of hardware and software system integration. Moreover, they typically integrate systems from multiple parties. Some of the integrated systems are even the same hardware boxes as the ones found on the real devices they replicate, an aircraft's cockpit display system being such an example in the case of a FMS. Furthermore, networking platforms together usually involves the interaction of disparate devices spanning across multiple integration sites. This results in a heterogeneous set of system interfaces which need to be interconnected together in order to deliver the platform's intended capabilities, training being an example for a FMS. The sole aspect

of the data exchanges is problematic ranging from data misalignment up to multi-architecture environments over varying kinds of communication protocols [1,2]. In this context, enabling system reuse across multiple platforms for product line support becomes even more challenging.

Therefore, this thesis contributes to the simplification of the integration of heterogeneous systems on platforms, such as FMSs, with the end goal being more cost-effective and efficient development, along with integration, of systems exhibiting better product line support.

Contributions - Automating System-Level Data-Interchange Software

The intent of this thesis is to facilitate system integration and system interoperability by automating the software required to connect systems together and to enable their interaction. To this end, this thesis proposes means to formally describe system interfaces from which the required software artifacts realizing the data exchanges can be derived and potentially be fully automated. Moreover, this thesis focuses on multi-architecture environments in order to facilitate the reuse of systems across platforms.

This is achieved with the System Interface Description Language (SIDL) which focuses on the data systems exchange and on the various aspects surrounding them. The primary focus of SIDL is the data systems exchange together. With system interfaces described in a formal language, it becomes possible to automate some of the tasks involved in achieving data interoperability, for instance, generating the software which deals with data serialization and protocol details, or which adapts a system interface to a prescribed unit of measurement (e.g., meters instead of feet).

Furthermore, having explicit system interface descriptions simplifies their validation, evolution and governance. That is because the proposed language used to describe them is a domain-specific language (DSL) thus boasts the vocabulary richness and expressiveness of a dedicated language describing system interfaces and their multiple facets.

System Interoperability Facets

SIDL is an architecture description language (ADL) as defined by [3] which is the ISO/IEC/IEEE International Standard for "Systems and software engineering — Architecture description". As such, SIDL is used to produce architecture description artifacts which formally capture the facets, or viewpoints, surrounding system interoperability.

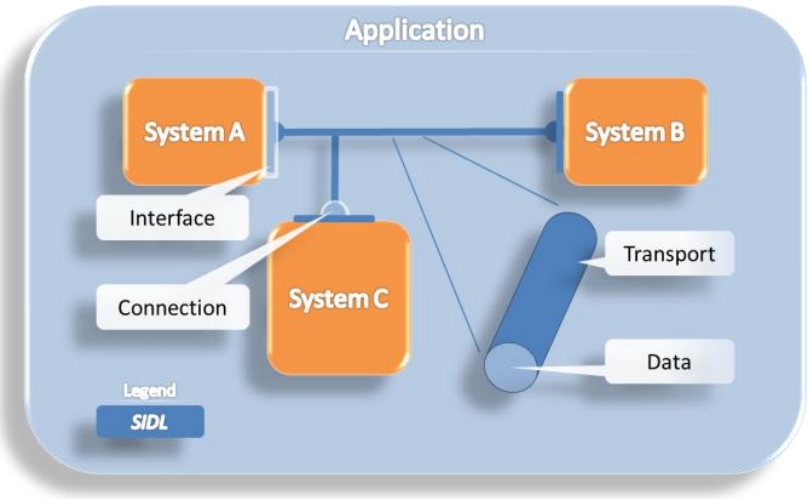

Figure 2: System Interoperability Facets

Conceptually, system interoperability revolves around four distinct facets as illustrated in Figure 2: the system's *Interface*, the *Connection* of the interface to data, the *Data* being exchanged between systems, and the data's *Transport* (e.g., protocol, middleware) from system to system. As an example, consider a computer with a USB keyboard as depicted in Figure 3. Both system *Interface*s, computer and keyboard, expose a USB port where the former inputs keyboard *Data* while the later outputs it. It is the *Connection* of each port together with a USB cable that enables the concrete *Transport* of the data from the keyboard to the computer.

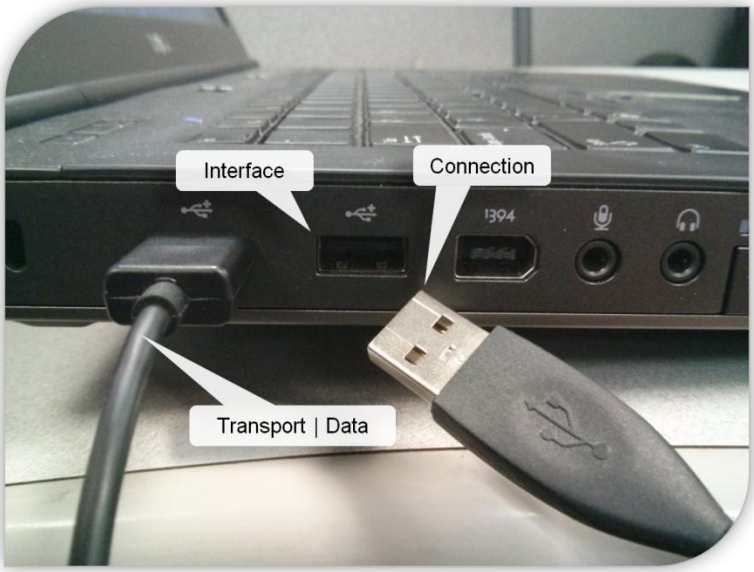

Figure 3: System Interoperability Facets Example

This thesis introduces the system interoperability facets as the core foundation used to partition and structure the problem domain as well as the proposed solution. The author selected this taxonomy, or organization, because each facet encapsulates a distinct interoperability area. This separation of concerns facilitates the understanding of this thesis and allows for a modular approach to system interoperability. Figure 2 illustrates these facets and highlights the scope of this thesis positioning SIDL accordingly.

Thesis Structure

This section introduced the subject of this thesis along with an overview of its contributions. The rest of the thesis is organized into three main parts. The *first* part provides the background information relating to system interoperability. It elaborates a current state of the art of the actual problems and solutions surrounding system interoperability covering its various aspects. It provides the rationale for a dedicated language aimed at describing system interfaces. Then follows the problem statement this thesis tries to address along with the research questions and objectives. The general approach consisting of generating software artifacts from formal descriptions of system interfaces is presented. The *second* part focuses on the methodology and the results obtained. It covers the System Interface Description Language along with examples using it. It also details test cases that demonstrate how the proposed methodology can enable better system interoperability and reuse. Finally, the *third* part presents a general discussion regarding the proposed methodology and its improvements over the current state of the art. It also concludes this thesis with a summary of the contributions made and points to potential ways forward that could improve upon this research. Following are the details regarding each part:

Part I - Introduction and Background

- Introduction: This is the introductory chapter of this thesis. It provides the background information required to understand this thesis. It also provides an overview of the contributions of this work.
- Chapter 1: This chapter presents an in-depth literature review covering both the problem domain and the existing solutions highlighting areas of improvement.

- Chapter 2: This chapter presents the general methodology covering problem statement, the research questions, as well as the research objectives that this thesis proposes to address. The general approach is also presented.

Part II - Methodology and Results

- Chapter 3: The System Interface Description Language (SIDL) elements are presented in this chapter. Each language element is described according to its relationship to the system interoperability facet that it relates to. Semantic rules are also presented covering the expected behavior of SIDL compilers. The expected usage of SIDL descriptions is detailed in the context of automating the system-level data-interchange software.
- Chapter 4: This chapter focuses on the experimental implementation used to create and validate SIDL. It covers the elaborated SIDL language, compiler, and code generator.
- Chapter 5: This chapter presents the experimental results of using SIDL to address specific test cases. Each test case is a distributed software application involving the interoperability of test systems. They are detailed with a particular regard over their data-interchange software which is generated from SIDL descriptions. It also covers the experiences of SMEs using SIDL in the development of the test systems.

Part III - Conclusions

- Chapter 6: This chapter presents a general discussion regarding the advances made by this thesis focusing on their implications and limitations.
- Conclusion: This is the concluding chapter of this thesis. It summarizes the key contributions made by this thesis and presents potential opportunities for future work.

Chapter 1 LITERATURE REVIEW

The delivery of capabilities over a specified platform often requires the integration of heterogeneous systems by an original equipment manufacturer (OEM). Moreover, from the perspective of an equipment supplier, the same equipment product line needs to deal with multiple integration platforms. In this context, this chapter focuses on understanding the span of the possible system integration issues related to data exchanges, as well as the issues impeding reuse and interoperability. This is achieved through an in-depth literature review covering both the problem domain and the existing solutions.

Moreover, the review focuses on two system integration perspectives: distributed simulations and platforms. The former perspective looks at the issues related to the integration of systems within distributed simulations such as training devices interacting together within the same training session. The latter perspective looks at the issues related to the integration of systems on a platform, for instance, the integration of avionics systems on a training device. From this perspective, the platform can also be seen as a system of systems.

The review also relates to the system interoperability facets which are illustrated in Figure 2. These include the system's *Interface*, the *Connection* of the interface to data, the *Data* being exchanged between systems, and the data's *Transport* (e.g., protocol, middleware) from system to system. These facets are used to structure the review since system interoperability conceptually revolves around them.

1.1 From Simulation to Distributed Simulations

Technology restricted to the domain of research and development, simulation is seen as the tool to model a platform and the environment in which it operates in the early 1970s [4]. During that period, advanced simulators have been successfully used in the design and engineering of new systems. It is with the improvement of simulators that the training community sees a marked interest in simulation as simulators are tailored to train civilian and military pilots. Unfortunately, [4] reports that this new type of training focuses only on the acquisition of skills to operate the simulated vehicle. The complexity required in training involving several aircraft at the same time

is so great that this type of training is only done on real planes. [4] explains that it was only during the mid-1970s that speculation began on the feasibility of distributed interactive simulations while the benefits of simulation began to be understood.

1.1.1 Simulator Network

In the early 1980s, it is generally recognized that the construction of a low-cost global networked military training system is virtually impossible [5]. It is a paper of Captain J. A. Thorpe, according to [5], which changes this by stating that it is not at the level of teaching techniques or at increased fidelity simulators to look out; Thorpe claims he should rather align training and actual combat systems to make them indistinguishable to minimize costs and maximize training. It is this statement which launches in 1983 an initiative dubbed SIMNET (Simulator Network) [4,5], to build a new generation of realistic distributed simulators at a cost one hundred times less than the existing generation that still does not allow the complex collective training involving several human-in-the-loop interactions. After more than 260 interconnected simulators across 11 sites in the United States and Europe, the prototype of SIMNET is seen to be a success. [5] added that the first SIMNET results of interconnecting worldwide simulators in real-time demonstrate the importance of practicing collective joint military exercises on a large scale under the same network infrastructure. Moreover, [5] reports that SIMNET even brings a new dimension to equipment acquisition practices with the advent of distributed simulations. Not only does SIMNET change the training industry, but it also changes how the military interacts with the industry by allowing for ready-to-use commercial off-the-shelf (COTS) components. This triggers billion dollars investment by the U.S. military to expand its global network of simulators for collective training and development of combat in the following years. These investments, according to [5], reflect the need for the Army to reduce costs through further industry involvement.

1.1.2 Distributed Interactive Simulation

In 1990, SIMNET change its name to Distributed Interactive Simulation (DIS) to eliminate the abuse of usage of the acronym SIMNET to denote any simulator network rather than the SIMNET distributed simulation [4]. In 1996, DIS becomes an IEEE standard [6]. It is the need to interoperate disparate distributed simulations which forces the development of standards

according to [7]. But [7] adds the equally important aspect of standards development is the need to expand business opportunities for simulator component suppliers. The advent of standards allows vendors to integrate their various simulation solutions under a single distributed simulation opening the door to new markets and meet the needs of targeted cost reduction by the U.S. military during the initial initiative with SIMNET (which is also confirmed by [5]).

1.1.3 High-Level Architecture

In the mid-1990s, the current simulation technologies as DIS did not achieve the Modeling & Simulation (M&S) vision of the U.S. Department of Defense (DoD) as reported by [8]. This M&S vision states the use of common environments such that the operations and acquisition domains be able to meet their respective responsibilities. Moreover, this vision requires that these M&S environments be constructed from affordable, reusable components interoperating through an open architecture. This is what drives a 1995 effort involving the public, private, and academic sectors for the development of a new distributed simulation environment in line with DoD's initial M&S vision, which is the High-Level Architecture (HLA).

According to [9], HLA is based on the premise that no simulation can satisfy all users and all possible uses, a premise which was not considered during the development of DIS. HLA is intended to be an interface specification rather than a specific implementation and wants to be programming language independent. After creating a reference implementation for HLA via its Defense Modeling and Simulation Office (DMSO), the DoD relies on third parties for providing commercial implementations and stops expanding its reference implementation always in pursuit of cost reductions. In 2000, HLA becomes an IEEE standard [10].

New traffic reduction techniques are required to enable the proper operation of large scale distributed simulations, as explained by [11], in order to meet their ever increasing network bandwidth needs. [9] explains that the services exposed by HLA's runtime infrastructure help manage the inherent complexity of communication protocols and thereby abstracting the use of the latest networking technologies. From [11,12,13,14], it is clear that HLA's exposed interface for managing data distribution drastically reduces the network traffic through the effective management of multicast groups.

From this point on, a number of other distributed simulation architectures emerged. These architectures are referenced accordingly throughout the following sections in the scope of system integration issues. The link from simulation to distributed simulation being established, we can now delve into distributed simulation interoperability.

1.2 Distributed Simulation Interoperability

A simulation, such as a flight simulation for example, is composed of objects that interact together. This is illustrated in Figure 1-1 where the objects represent aircrafts. Within this *object model*, objects interact with each other, among other things, through the information, or *Data* (Figure 2), they exchange.

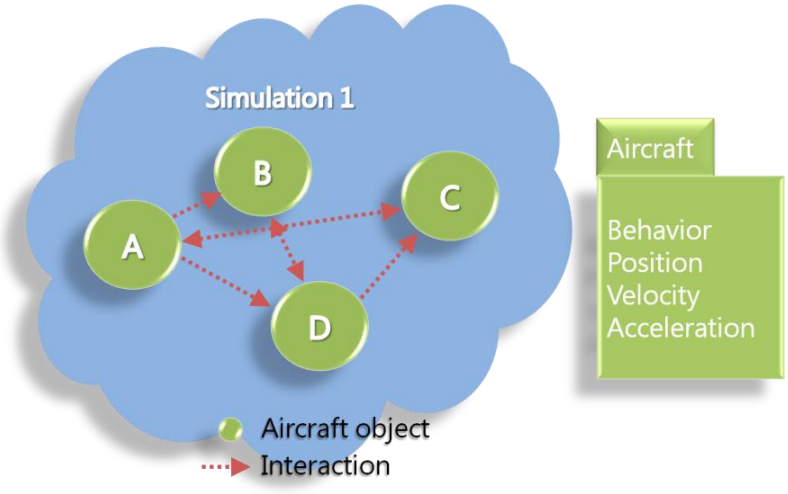

Figure 1-1: Non-Distributed Simulation

The *behavior*, *position*, *speed*, and *acceleration* of the aircrafts in Figure 1-1 are examples of the *Data* enabling an aircraft pilot to train with other virtual aircrafts. To enable the training of several pilots in the same training session, different simulators have to be *Connected* (Figure 2) together. The interconnection of multiple simulations requires a network exposing a communications protocol which allows for the same interactions as when there is a single simulation. That is what Figure 1-2 illustrates by presenting a distributed simulation.

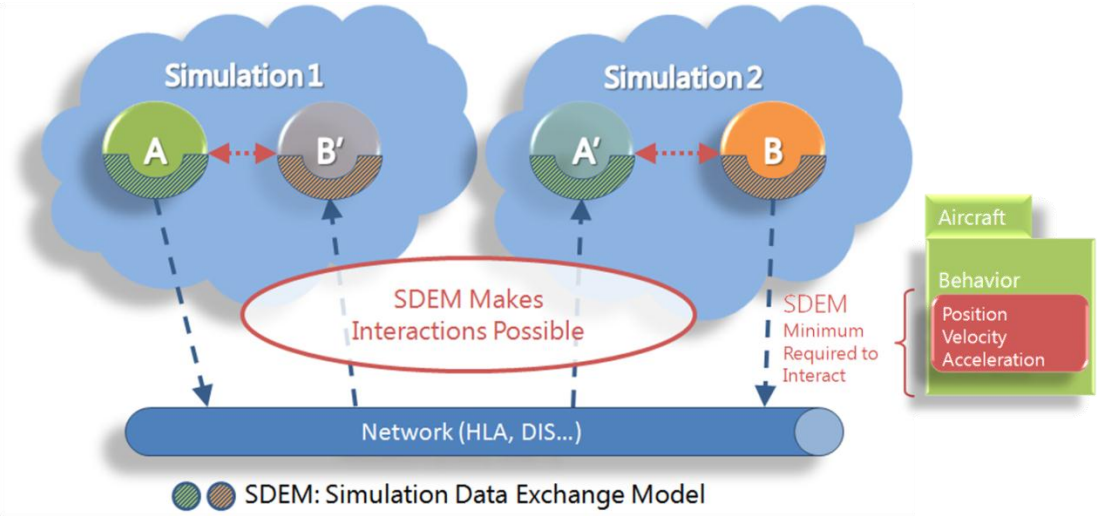

Figure 1-2: Distributed Simulation

Excluding communication protocol, or *Transport* (Figure 2), peculiarities, a fundamental trait of distributed simulations is that only a subset of the available *Data* is required to enable the same interactions as when the simulation is not distributed. This minimum set of data is known as the Simulation Data Exchange Model (SDEM). As illustrated in Figure 1-2, this minimum set is composed of the aircraft's *position*, *speed*, and *acceleration*, the *behavior* not being required to enable distributed interactions.

It is the need to interconnect different types of distributed simulations that gave rise to the concept of interoperability and appeared from the very beginning of SIMNET [7]. Interoperability, as defined by [15] is "the transfer of information that preserves the meaning and relationships of the information exchanged". Typically, according to [16], a gateway is used for distributed simulations interoperability such that the whole is seen as a single, unified, simulation. Figure 1-3 presents this *interoperability agent* which bridges between different distributed simulations. Here, the agent adapts the information of two simulation data exchange models, notably *SDEM 1* and *SDEM 2*.

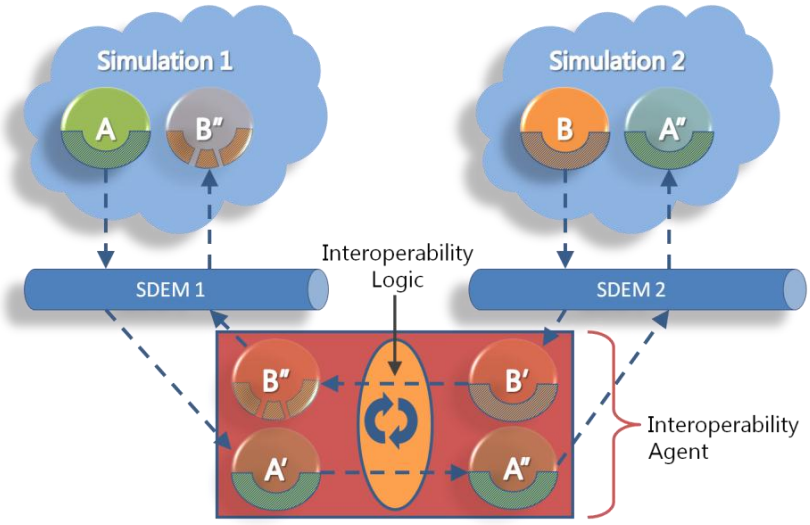

Figure 1-3: Achieving Interoperability through an Interoperability Agent

Two solutions, as pointed out by [16], are typically used to interoperate distributed simulations: adapt the existing applications by modifying their source code, or develop a gateway, i.e., a software bridge, to fulfill the interoperability needs. The adaptation of existing applications may be required for technical reasons, such as latency or throughput constraints, or for necessary migrations as articulated by [13,14,17,18]. Except that it is not always possible either for technical reasons, such as accreditation or security considerations, or simply for economic reasons as demonstrated by [19,20,21]. That is why most interoperability problems are solved using a gateway which is even qualified as a "necessary evil" by [15].

Moreover, it is not always possible to preserve all the *Data* exchanged. That is the case for the object B' which is transformed into B" (Figure 1-3) since a subset of the information has no correspondence in *Simulation 1*. To preserve the integrity of the *Data* exchanged as much as possible, the interoperability logic has to face such eventuality which significantly increases its complexity. Interoperability agent development often requires case-by-case development because of the intrinsic characteristics of the distributed simulations to interoperate, and involves multi-disciplinary teams.

1.3 System Interoperability

A distributed simulation is populated by systems which provide its content. For instance, a training device is such a system because it exposes a virtual representation of the aircraft it

replicates within the distributed simulation. This implies that the distributed simulation is the artifice which enables systems to interact together. Therefore, “distributed simulation interoperability” effectively refers to system interoperability. On a larger scale, a distributed simulation can be seen as a system itself in which case a gateway effectively allows systems, i.e., distributed simulations, to interoperate. That is why this thesis generalizes the concept of distributed simulation interoperability to system interoperability as emphasised by the *Interface* facet of Figure 2. Even though distributed simulations exhibit distinct integration issues, they are seen from the perspective of system interoperability throughout this thesis.

1.4 Multi-Architecture Environments

Modern system integration scenarios typically involve multi-architecture environments. That is, the systems being integrated together expose interfaces that are not aligned over a single architecture. Multi-architecture case studies are presented by [1] primarily in the context of Live Virtual Constructive (LVC) environments. These environments integrate various distributed simulation architectures such as DIS, HLA, and the Test and Training Enabling Architecture (TENA). In order to assist the development and the execution of such applications, the authors propose an overlay to the IEEE Distributed Simulation Engineering and Execution Process (DSEEP) [22]. This overlay targets specific activities that exhibit multi-architecture issues. The following sections dig into these issues.

1.5 Meta-Model Incompatibilities

Each architecture essentially has its own meta-model, or way of representing the data being exchanged. The data being exchanged forms a model which is captured by a meta-model. Possible meta-model incompatibilities are highlighted by [1] when the simulation data exchange model (SDEM) is being developed. The authors ideally propose that the SDEM be developed in an architecture-agnostic way. This would ensure that the semantic meanings of the data representation in each SDEM be preserved across each architecture. For the time being, they recommend the use of gateways due to the lack of an architecture neutral meta-model.

Meta-model incompatibilities are the concern of the *Data* and *Transport* facets (Figure 2). Because the meta-model deals with how data is represented within an architecture, i.e., the

primitives used to model the data, how the data is serialized, the data's quality of service (QoS) such as reliability and latency budget.

1.6 Data Incompatibilities

Another multi-architecture issue listed by the authors of [1] is SDEM data incompatibilities. The data that systems exchange needs to have equivalent semantics across all architectures for interoperability to occur. A *position* concept, for instance, needs to be uniformly represented with the same units and frame of reference. The authors propose to either re-align (i.e. refactor) systems, which has the greatest cost, or use gateways to bridge the gaps. They even recommend architecture-agnostic gateways.

Data incompatibilities are primarily the concern of the *Data* facet because they directly impact the data itself. But they also touch the system's *Interface*, since systems exchange *Data* through it, and the *Connection* as that is how the system's *Interface* is connected to the *Data*. Data incompatibilities arise when trying to connect an *Interface* that uses a different unit of measurement or frame of reference for instance.

1.7 Lack of Architecture Neutral Meta-Model

The lack of an architecture-agnostic way of expressing data exchange models slows the whole integration process as highlighted by [2]. It renders the mapping between training and experimental objectives, along with the data exchange models supporting them, more challenging and prevents further automation. The authors expose early work on an Architecture Neutral Data Exchange Model (ANDEM) trying to represent HLA, TENA, DIS and CTIA (Common Training Instrumentation Architecture) data models into a core one with specific architecture mappings. The authors also studied the Base Object Model (BOM) template specification [23]. They point out that BOM can be used to map high-level conceptual models to data exchange models. But as BOM is focused around HLA, this needs to be generalized. The follow-ups to this work are the SISO ANDEM Study Group and SISO BOM specification revision which aim at addressing these issues.

Other early work in this field is presented by [24] which suggests a Neutral SDEM format along the lines of [25]. That is a SDEM whose format is not associated with any distributed simulation

architecture. This ongoing research primarily targets the simplification of gateways which translate between SDEMs.

1.7.1 The FACE Technical Standard

The Future Airborne Capability Environment (FACE) Technical Standard [26] proposes its own meta-model in the context of warfighting platform development. The general approach used by FACE, as highlighted by [27], is to develop a standard for a software computing environment designed to promote software product lines across different platforms therefore enabling increased reuse. Ultimately, the goal of FACE is to reduce development and integration costs and reduce time to field new avionics capabilities. Therefore, FACE proposes a data modeling methodology covering platform-independent and portable models down to platform-specific ones. These models are separated into Conceptual, Logical and Platform data models, respectively, each level refining the previous one with the latter being platform-specific. In this context, this thesis generalizes the notion of SDEM, which only focuses around simulations, to that of data exchange model (DEM). This allows platforms and distributed simulations to be seen from a common perspective. The FACE standard also deals with *Interface*s through units of portability which package full services or mission-level capabilities to systems.

The main problem with FACE's meta-model is that it does not capture the details of each individual architecture realizing the models. That is because FACE standardizes a technology stack specifying details down to the Operating System (OS). Hence, the meta-model does not need to deal with multi-architecture issues from the perspective of a FACE-compliant platform. In fact, the meta-model only allows for data variability by providing data conversion and aliasing mechanisms. Except issues arise as soon as a non-FACE component gets into the picture. From the perspective of an equipment supplier, the same equipment product line needs to deal with FACE and non-FACE platforms nevertheless. Another example is performing a joint exercise with non-FACE platforms which invariably confronts against multi-architecture issues.

1.7.2 Web Services Similarity

A web service is a method of communication between electronic devices over the web. The Web Services Description Language (WSDL) is an XML format for describing web services [28]. It is commonly used in large-scale distributed applications particularly with ones based on the

Service-Oriented Architecture (SOA). WSDL addresses the need of communication formalization for web services and formalizes the set of operations services expose as architecture-neutral service interfaces.

1.7.3 Service versus System

Conceptually, a web service is very similar to a system part of a platform (or a distributed simulation). Both revolve around the *Transport* of *Data* through an explicit *Interface*. The main disparity between the two originates from the intent of their interfaces. A system can only interact with other systems if its *Interface* is aligned with them, that is if it can be *connected* to the available *Data*. A service, on the other hand, is only concerned with making its *Interface* available to its consumers. That is why a service contract is required to capture the interactions between service providers and consumers as described by [29]. That is why WSDL does not cover the *Connection* facet and which must be captured to enable system interoperability. On the other hand, WSDL allows capturing a service's expected interaction sequences which ensures coherent service behavior. This is achieved through WSDL's message exchange patterns (MEPs). Nonetheless, there is a natural fit between the services and systems as they share the other facets.

1.8 Representing Data Exchange Models

Data exchange models represent *what* is being exchanged between systems. There exists a multitude of ways for capturing them. The following sections describe some of these ways highlighting issues, along with limitations, related to capturing *Data* explicitly.

1.8.1 Lack of Machine-Processable Definitions

Some data exchange models are represented as paper specifications. DIS is such an example and describes *Data* through various textual descriptions and tables [6] which are aimed towards implementers. It is sufficient to say that paper specifications prevent any form of automation. Moreover, in order to extend capabilities not initially covered by DIS, open constructs are provided which producers and consumers need to agree on. This implies that these extensions are captured in varying ways and on a case-by-case basis being left outside the scope of the specification.

Another issue is the presence of definitions within SDEMs which prevent automation by a software agent. This is particularly problematic when trying to automate the serialization of an SDEM through code generation. These types of definitions render code generation choices complex and even un-automatable requiring manual intervention. An example of this is the *encoding* attribute of HLA data models which is an open text field [17]. It can be filled with a pre-defined encoding type in which case the automation is not problematic. Except *encoding* is sometimes defined as the set of instructions a developer is expected to read in order to handle the serialization manually. Such definitions need to be captured in a form entirely processable by machine as well as exempt from ambiguity and misinterpretation.

1.8.2 XML-Based Format Deficiencies

Most data exchange models are typically captured explicitly in a machine-processable format. That is the case of the HLA Object Model Template (OMT) standard [17] which provides an XML format to capture HLA data models. OMT specifies the XML schema [30] used to define the data available to HLA distributed simulations. The OMT schema provides some form of validation which XML tools and runtime libraries can use to produce or consume the data models. For instance, an OMT rule is defined to ensure that a referenced type is defined in the SDEM. This form of rule is simple to capture with XML Schema. Unfortunately, not all XML tools and libraries validate these rules, and only a few cover the full spectrum of XML Schema's validation capabilities.

Moreover, complex rules cannot be expressed with XML Schema as it focuses around structural validation. For instance, it is not allowed with OMT to have sibling subclasses with identical names, or to have user defined type names which start with the letters "HLA". Supporting such rules could require modification of the schema which is impractical as XML is primarily meant to be an interchange format. Therefore, one needs to balance between simplicity in validating the format, and ease in producing or consuming it. Nonetheless, OMT is a modular format enabling reuse and extension of existing data models in a guided way. For interoperability to occur, systems must share the same SDEM. Failing to do so require adapting the unaligned data models and is left out of the scope of the OMT specification.

Another XML format used to represent SDEMs is the BOM standard [23]. It was a response to the need to increase the level of abstraction of OMT by introducing conceptual modeling

capabilities through its patterns of interplay. Being built around HLA currently prevents BOM from being used to represent other types of SDEMs. Nevertheless, BOM's ability to model patterns of communication exhibits great value capturing dynamic data exchange details in an architecture-agnostic way.

WSDL [28] is a standardized XML format used to describe web services. It requires that an external type system be used in order to describe the messages passed between a service and its consumers. XML Schema typically fulfills this purpose. The notion of external type system is interesting, except it creates a burden on WSDL consumers, such as service frameworks, which need to cope for type system limitations, such as with XML Schema, and variability. As highlighted by [31], WSDL lacks the real ability to fully capture service data models because of this. Even so, WSDL provides Message Exchange Patterns (MEPs) similar to BOM's patterns of interplay which capture the dynamic details of data exchanges.

In the context of ongoing research, the authors of [24] propose an architecture-neutral XML Schema-based format to represent SDEMs. They identify the key characteristics of SDEMs which consists in a format, a data structure, and semantics. The format captures how to represent the information; the data structure describes the content in the specified format; while the semantics provide meta-information about the data model. The notion of a format agnostic to architecture as proposed by the authors is interesting, except the authors limit their research to HLA data models and focus on describing mappings between SDEMs in order to describe gateways. Alternatively, the authors of [2] are looking towards the semantic web, such as ontologies, for added semantic capabilities on top of XML in order to capture SDEMs in an architecture-neutral way. The authors describe how ontologies could be used for this purpose, but no concrete detail is given as their work is too preliminary. One point emphasized is the need for human readability and machine understanding. It is important to point out that these requirements are generally incompatible, one impeding the other.

1.8.3 Language-Based Format Deficiencies

The Interface Definition Language (IDL) [32] is a standardized language used to describe the interfaces of software components independently from the languages used to implement and use them. IDL requires a compiler which not only validates IDL definitions, but also transforms them into other forms. There even exists standard language mappings such as IDL to C++ [33]. This

enables IDL to provide strong validation constructs over data models with consistent behavior. Amongst its users, Data Distribution Service (DDS) [34] middlewares use a subset of IDL to represent data exchange models. Being a dedicated language enables IDL to provide a clutter-free view of the data models it captures. DDS uses IDL to automate the data serialization software through code generation. Unfortunately, IDL only captures the low-level details of data models being close to their computing platform equivalent. For instance, IDL data types allow the representation of integer and floating point numbers, but lack the capability to qualify data with engineering units of measurement or frames of reference. Additionally, some data models, such as DIS and HLA OMT, require explicit enumeration values which IDL forbids. This prevents IDL from being used to completely capture these types of data models.

The TENA Definition Language (TDL) [35] is similar to IDL which it actually extends to meet specific use cases (such as considering local versus remote objects). Besides exhibiting the same characteristics mentioned above, TDL enumeration identifiers need to be unique and vector cardinalities cannot be bounded to represent fixed-length arrays [36]. These restrictions simplify the TENA middleware, except they prevent capturing some types of data models. Nonetheless, it allows TENA to generate distributed applications from TDL descriptions as noted by [37]. Except interoperating with other architectures still requires gateways which cannot be automated.

The Abstract Syntax Notation One (ASN.1) [38] is a standard notation for describing data along with the rules for serializing and transmitting it. ASN.1 is principally used in the telecommunications industry. It shares many traits with IDL by providing an abstract syntax independent from the languages used to implement and use it [39]. Its type system accounts for bit representations as one of its primary purposes is to provide compact binary representations. In regards to system integration, the principal issue with ASN.1 is that it is aimed towards protocol designers [39]. Moreover, its notation differs in many ways to traditional programming languages, or from languages such as IDL, which makes ASN.1 less accessible. Additionally, as opposed to IDL, ASN.1 does not provide guidance on mapping it to programming languages since it focuses solely on uniform encoding behaviors. It also lacks higher-level abstractions, such as units of measurement and frames of reference, which typically hamper interoperability.

Conversely, the Architecture Analysis & Design Language (AADL) is an architecture description language used to model software and hardware architectures of embedded real-time systems [40].

AADL represents data through property type declarations which provide a set of values with characteristics such as units of measurement. Because it is primarily aimed towards analysing architectures, there exists many ways and scenarios to represent data. This makes AADL awkward to represent data in the context of describing data exchanges. Moreover, the AADL grammar is challenging to read even if textual.

1.8.4 Model-Based Format Deficiencies

On the other hand, FACE proposes a model-driven methodology based around UML modeling. Its three-levels of *Data* modeling through refinement (i.e., conceptual, logical, and platform) enable strong data semantics and cover many reuse scenarios. It also accounts for a uniform mapping from UML models to IDL, as covered by [26], which enables consistent software artifact generation across FACE platforms and supported programming languages. Unfortunately, modeling is cumbersome and very repetitive simply to create the required artifacts because each element needs to be modeled three times. Another model-based format is the Systems Modeling Language (SysML) which is a general-purpose language for system modeling [41]. SysML proposes a UML profile, i.e., a subset of UML with extensions, which is meant to be customized in order to create domain-specific modeling languages such as for the automotive and aerospace domains.

Unfortunately, the lack of model validity feedback from UML editors is a concern noted by [42] and impacts both FACE and SysML. The authors also expose scalability problems of UML tools particularly when dealing with large modeling environments seeing load times exceeding one hour in some cases. Furthermore, they point out that UML lacks expressiveness making it challenging to capture information in a natural way.

Additionally, UML models are often persisted in proprietary formats although they can be exported using the XML Metadata Interchange (XMI) format [43]. Being a XML format, XMI therefore exhibits the same limitations as the ones presented in Section 1.8.2. Another caveat with XMI is that most UML tools do not interoperate well even if XMI and UML are standards, as exposed by [44,45], thus undermining reuse. FACE's IDL heritage also causes the manifestation of the issues elicited in Section 1.8.3 when using the UML to IDL mapping.

1.8.5 Data Type Deficiencies

Data exchange models capture data of varying representations. Most provide data representations, i.e., data types, aligned with computing platforms. These include representations of numbers, strings, characters, Booleans, and enumerations. Also available are structures composed of fields which in turn are of a particular data type. Arrays are found amongst data types and are used to represent a bounded or unbounded set of values of the same type. Unions, or variants, are often encountered which are used to represent a value which has a finite set of varying forms such as for a uniform array of different items.

HLA OMT can express all of the above data types with the exception of unsigned numerical types [17] which are required when representing positive numbers. Likewise, XML Schema, IDL, and FACE prevent the association of literal values to enumerators within enumerations [26,30,32]. Another issue with FACE is that it lacks variant type support [26] which is also the case for AADL [40]. These issues prevent completely capturing existing data models such as DIS and HLA OMT.

1.9 Lack of Transport Details

Capturing data exchange models explicitly only addresses the *Data* facet covering *what* is exchanged between systems. The ability to capture *how* data is transported between systems is covered by some data exchange model representations. Some representations, such as IDL, do not cover *Transport* details at all focusing solely on *Data*. Conversely, some formats cover many *Transport* aspects, except they lack some of the details required to enable system interoperability. As an example, the author of [31] presses that WSDL fails to capture non-functional service characteristics, such as quality of service (QoS), which is the root cause of several service interoperability problems.

Alternatively, ASN.1 covers *Transport* details by providing reusable means to model how data is encoded which, in turn, can be changed independently from the abstract data representations. This separation of *Data* and *Transport* is also present in WSDL which enables specific protocol characteristics to be captured in bindings even if QoS coverage is deficient. Regrettably, ASN.1 protocol designers cannot control the final encoding which is often required in order to interoperate with existing systems [46].

Transport details are not captured within AADL models per se, except AADL provides the capability to represent a communication link through its conceptual bus construct [40]. For instance, representing an HLA transport would require creating an HLA bus. Moreover, some predefined general transport characteristics can be captured through AADL properties. Unfortunately, these are quite limiting and prevent AADL from capturing the sought transport details.

Another issue is capturing *where* data is accessible from. One could argue that this is a concern of the *Connection* facet, except the *Connection* facet is concerned with how systems are logically connected to one another whereas the *Transport* facet is concerned with the concrete details required to realize data exchanges. This is also consistent with WSDL which considers *Transport* to be concrete [28]. WSDL provides access to services through endpoints which capture *where* a service is accessible from. Moreover, it represents *how* data is transported between the service and its clients through bindings which capture the message format and transmission protocol. Bindings enable WSDL to consider multi-architecture environments by capturing the peculiarities of each architecture in a distinct binding.

1.9.1 Intermingling Transport with Data

Some data exchange model representations mix *Transport* directly on *Data*. This prevents data reuse within the same architecture and across multiple ones. For instance, *Transport* attributes, such as encoding and QoS, are expressed directly on HLA *Data* [17]. Consequently, reusing HLA *Data* can only be achieved within HLA architectures. Furthermore, this can only be realized by duplicating and by adapting the *Data* to the various *Transport* use cases. Transposing the same data over multiple architectures further duplicates the information. Therefore, there must be a clear separation between *Data* and *Transport* as captured within a data exchange model representation in order to enable reuse.

1.10 Lack of Interface Details

Most data exchange model representations only focus on capturing *Data* with some detailing *Transport* characteristics. Except, in order to simplify system integration, one also needs to represent systems as they are the focal point of the integration process. One typical type of integration issue relates to systems with diverging interfaces. Consider two data types with

different units of measurement. From the *Data* facet's perspective, these two types provide no indication of possible integration issues. Except systems requiring aligning these two types together proves otherwise if the systems need to interoperate together. Therefore, an explicit description of each system's *Interface* is required in order to capture, and identify, these potential issues.

Service interfaces can be captured with WSDL [28] where the operations supported by a service are detailed on an interface element. Interface elements then relate to *Data* by specifying the inbound and outbound message types. This enables WSDL to fully describe web services covering the *Interface*, *Data*, and *Transport* facets.

Conversely, AADL proposes an abstraction of components accounting for application software, execution platform, and composite components [40]. As an example, AADL can represent software processes, hardware processors, and memory components. One of the central AADL component is the system component which is used to model distinct units within an architecture. System components expose ports enabling them to exchange *Data* together. As with WSDL, AADL values the explicit capture of system interfaces. To this end, [47] uses AADL to facilitate middleware analysis by formally capturing the complete systems to interoperate and by describing the required behavior of the middleware. The problem with the proposed approach is that it only works with their architecture as AADL does not model transport which their architecture compensates for.

As with AADL, FACE proposes a way of capturing system interfaces through its Unit of Portability (UoP) [26]. Each FACE UoP is composed of ports which makeup its *Interface*. These definitions can then be used to automate some of the data exchange software through code generation. In a similar fashion to FACE, SysML proposes a native *Interface* element which can be composed of ports. Therefore, the ability to formally capture system interfaces is key in describing data exchanges and enabling automation.

1.11 Lack of Connection Details

Capturing the *Interface* of systems, the *Data* they exchange, and how this data is *Transported* does not account for a full description of system data exchanges. One needs to know how data is routed between systems, that is, how systems are connected to one another. This is the purpose of

the *Connection* facet. Unfortunately, of all the languages explored, only SysML and AADL capture system connections by modeling the connections between system ports. Moreover, the lack of such details renders connections implicit and subject to interpretation by SMEs. This can cause an SME to input data from the wrong source. These types of issues are quite challenging to detect and are often found late into the integration phase.

1.12 Subject Matter Expert Modeling Complexity

One of the main problems with the existing solutions is the required knowledge to use them which is not typically shared amongst subject matter experts (SMEs) and impedes productivity [42,48] which, in turn, directly impacts system development and integration. As an example, [24] justifies the use of XML for capturing data exchange models based on the fact that it provides human readability, machine readability, and existing tool support. Being a human-readable format does not necessarily make it human-understandable too. This explains why [31] stipulates for WSDL, an XML format, that "[WSDL] descriptions are machine-readable rather than human-friendly".

Another example involves SMEs which are expected to directly translate their domain expertise into a software form [48]. As software programming increases in complexity, there is a conflicting duality of requiring SMEs to be experts both in their domain and in software. In the same lines, HLA OMT exposes system experts to protocol details by intermingling *Transport* with *Data*. Except the SMEs' primary concern regards consuming and producing system data. On the other hand, system integrators are concerned with such details.

Languages, such as IDL and ASN.1, are more restrictive exposing dedicated viewpoints aimed towards specific SMEs thus hiding complexity from them. Unfortunately, these languages lack the level of abstraction expected by SMEs focusing solely on a first-degree abstraction from computing platforms and software protocols. For instance, units of measurement and frames of reference are not an integral part of IDL. This forces SMEs to deal directly with their computing platform representations instead. To cope for this, an SME could model an IDL `Altitude` type as an altitude `Value` with its `Units` and `Frame` of reference representations as follows:

```
enum AltitudeUnit
{
  Meter,
  Feet
};
```

```
enum AltitudeFrame
{
  AboveGround,
  EarthCenter
};
struct Altitude
{
  double Value;
  AltitudeUnit Units;
  AltitudeFrame Frame;
};
```

Except the `Altitude` type is no longer a single 64-bit `double` precision floating-point `Value` from IDL's perspective therefore impacting its computing platform representation and transmission size. Additionally, an IDL compiler cannot help SMEs with units and frame errors as it does not know about them. In opposition, XML formats cope for this by providing richer representations. Moreover, XML formats require to be transformed thus eliminate the superfluous information not required by the computing platform representation.

In contrast, UML provides the capability to create profiles, which are similar to dedicated languages, exposing rich constructs and abstractions therefore hiding complexity from SMEs. FACE uses this capability, as presented by [26], as well as SysML, to combine the aforementioned benefits of both the dedicated languages and the XML formats. Except UML modeling exhibits many pitfalls when trying to create dedicated languages, i.e., profiles, as uncovered by [49] which states the lack of framework support, existing tool limitations, and a complex UML extension mechanism. The lack of expressiveness is also noted by [42] which proposes domain-specific languages (DSLs) instead. That is because a DSL boasts the vocabulary richness and expressiveness of a dedicated language aligned with an SME's domain.

1.12.1 Leveraging SME Expertise

The main purpose of a full mission simulator (FMS) is to provide training to aircraft crews. From a software application perspective, replicating an existing aircraft and its environment involves the interaction of several systems within the FMS (Figure 1). Each system has its own set of simulation models, for instance, aircraft system models, such as flight and engines, and models dealing with the environment surrounding the aircraft, such as weather and air traffic. In addition, multiple FMSs can be joined together to simulate air traffic. Moreover, system models need to communicate data to other systems, thus demonstrating particular data interests, in order to replicate an end function of the aircraft to the pilot. In a nutshell, this technology-agnostic description represents *What* (Figure 1-4) SMEs need to build to achieve an FMS. Additionally,

because SMEs are experts in a particular domain, abstractions need to be available to them so they can use resources, such as the hardware, without requiring being experts in these resources.

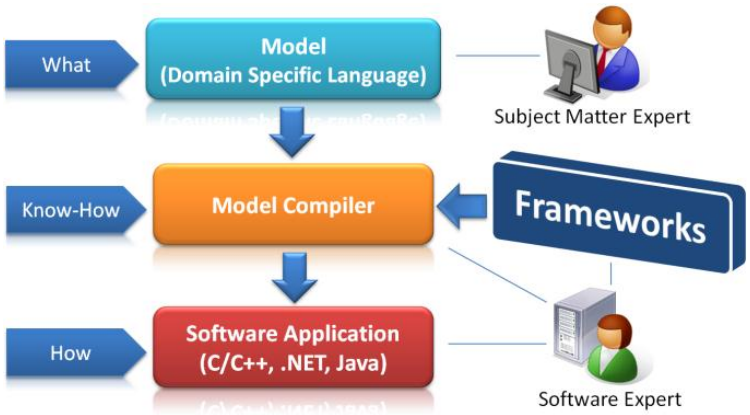

Figure 1-4: Hiding Software Complexity from SMEs

Next, *How* to actually implement an FMS involves many different kinds of expertise covering the whole application's scope (hardware, software, OS, etc.). Moreover, reusable components provided by frameworks can be composed together to form the software application complemented by model-specific user code and parameterizations. This software application is also subject to multiple configuration parameters and is deployed on hosts interconnected via diverse networks. Moreover, because system models elicit their communication's quality of service (QoS) requirements such as data transfer latency, deployment strategies need to be considered. For instance, if the expected transfer latency is high, then a strategy involving network communication might be appropriate. If the latency is low, then system models communicating together might be required to run on the same host or in the same process. Furthermore, with a 64-bit OS capable of accessing large amounts of memory, the number of simulation models per process is likely to increase in order to leverage multi-core CPUs therefore it becomes important to efficiently address communications within the same process. This unfolding of the *What* into the *How* requires *Know-How* (Figure 1-4) that is not shared by the majority of SMEs. As such, there is a need to abstract this complexity in order to leverage the expertise of SMEs.

1.12.2 Hardware Performance

Another source of complexity to hide from SMEs is the alignment with hardware. Indeed, SMEs tend to model from abstractions that are close to their corresponding real-world equivalent. For

instance, one would model a list of aircraft as an *Array of Structure* (AoS) (Figure 1-5). However, in order to fully leverage today's hardware capabilities, a specific memory layout must be used by models to efficiently move data in/out of the CPU and to demonstrate the capability to simultaneously perform multiple computations which Brownsword [50] and Collin [51] highlight. For instance, the Single Instruction, Multiple Data (SIMD) instructions [52] supported by modern multi-core CPUs and GPUs process a great deal of data in parallel. An example of an efficient hardware model is a list of aircraft data structures represented as a *Structure of Array* (SoA) (Figure 1-5).

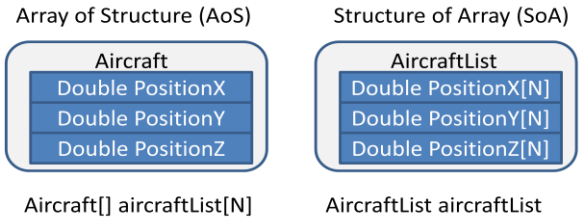

Figure 1-5: Array of Structure vs. Structure of Array

To illustrate the impact of the data memory layout strategy on performance, following AoS and SoA, we computed the time taken to execute the following kinematic equations on a list of aircraft data structures considering their position, velocity and acceleration:

$$p_{i+1} = p_i + v_i t + 0.5at^2 \text{ and } v_{i+1} = v_i + \text{ at.}$$

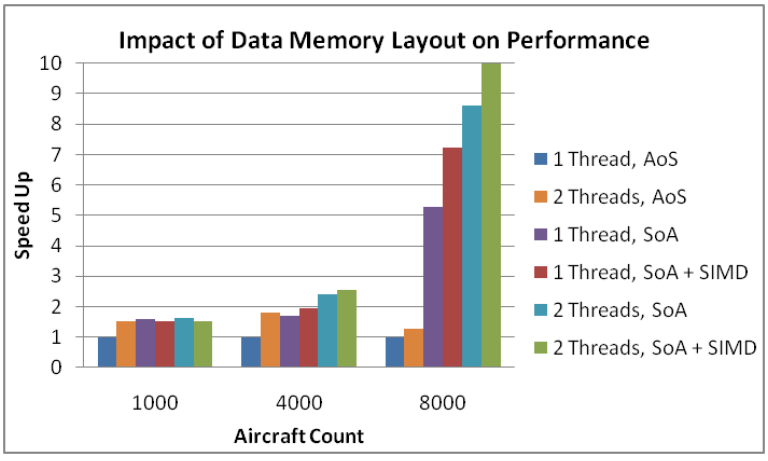

Figure 1-6: Impact of Data Memory Layout on Performance

As shown in Figure 1-6, using a SoA memory layout has significant impact on the speedup when compared to a single-threaded execution using an AoS memory layout. Moreover, a SoA

memory layout allows the use of SIMD instructions and improves the speedup gains of using multi-threaded execution. Again the *Know-How* (Figure 1-4) required to store a simulation model's data (i.e., *What*) into a hardware efficient data memory layout (i.e., *How*) is not shared by the majority of SMEs. Moreover, such a data memory layout depends on the actual computations applied to the simulation model. Consequently, each simulation model's data memory layout may differ, which leads to suboptimal hardware performance. As such, there is a need to abstract this complexity with considerations at the level of the whole software application.

1.12.3 Frameworks

Frameworks are a further source of complexity for SMEs because they offer too much latitude in the way they can be used [53,54]. Frameworks are programmed using a general-purpose language, and if SMEs have access to the full expressivity of the language, this increases the probability that they can introduce software defects, create suboptimal solutions or apply non-uniform solutions to a recurring problem due to their lack of software expertise. SMEs could also make technological choices directly within their code that others would have no knowledge of, which can cause problems with managing technology obsolescence and evolution. Imagine the difficulty of managing the network resources of an FMS when systems exchange data through the network without exposing that resource usage.

1.12.4 Domain-Specific Languages

A domain-specific language (DSL) allows SMEs to focus on *What* (Figure 1-4) needs to be done while abstracting the complexity of having to specify the full algorithmic details needed to implement a software application that does the *How* [55].

An example of a DSL is the Structured Query Language (SQL) designed, among other things, to insert, update, delete and query data in a relational database. An SQL query allows the user to describe its data interest, the *What*, while avoiding the need to describe the necessary operations to produce the expected results, i.e., the *How*. For instance, an SQL query to retrieve Books with a price greater than 100$ would be:

```
SELECT * FROM Book WHERE price > 100.00
```

As such, this query doesn't specify the details of the operations needed to locate, retrieve and filter the expected Books from a persistent storage. This gap between the *What* and the *How* provides users with a simplified experience and gives relational database implementers leeway for specific optimizations.

In contrast to a general-purpose language used to create a framework, a DSL is a computer programming language of limited expressivity that focuses on the core concepts and behaviors of a particular domain. Therefore, SMEs working with a DSL are more restricted in the valid programs they can create than they would be with a framework [53].

1.12.5 Model Compilers

Following the analogy of a compiler transforming a high-level language such as C++ into a low-level assembly language, a *Model Compiler* (Figure 1-4) contains the software knowledge *Know-How* needed to transform a model created using DSLs into a software application. Since models created using DSLs are domain specific, compiling them also requires a domain specific *Model Compiler*. More specifically, a *Model Compiler* generates from a model part or all of the software assets, like the C++ user code to complement a framework's reusable components and an application's configuration files, needed to obtain an FMS software application. For instance, a *Model Compiler* can generate all the C++ code needed to achieve data-level interoperability [56] between two simulation models for a particular data format and communication protocol. Also, using a *Model Compiler* to generate software assets reduces the risk of introducing software defects, creating suboptimal solutions or applying non-uniform patterns to a given recurring problem.

Moreover, a *Model Compiler* can deal with the QoS requirements of communications such as data transfer latency. Based on the expected data transfer latency between two simulation models, a *Model Compiler* can opt to deploy the models separately on different computing nodes or to combine them in the same process.

Again, similar to a compiler generating assembly code from C++ code but allowing debugging at the higher-level C++ abstraction, a *Model Compiler* can generate the code needed to support debugging using the level of abstraction of DSLs thus hiding complexity from SMEs. In addition,

the gap between the *What* and the *How* provides potential for a *Model Compiler* to apply context specific optimizations. An example of such optimization would be to transform a list of aircraft modelled from an *Array of Structure* (AoS) into a list of aircraft stored in memory as a *Structure of Array* (SoA) (Figure 1-5).

1.12.6 Code Generation

Code generation is used by existing solutions principally to automate software artifacts such as source code and configuration data. For instance, FACE and DDS use IDL to automate data serialization [26,34]. Another example is TENA which uses compiled-in object definitions [57]. The author noted that code generation is used by TENA because it provides strong type validation, detects errors early, enables better performances, and it conforms to current best software engineering practices. AADL also enables code generation as demonstrated by [47] for middleware analysis.

Code generation approaches, particularly the ones using model compilers from DSL models, offer advantages sought by the approach of this research. First, simplifying the task of software development by automating some code allows SMEs to limit their work on the problems they are trying to solve rather than having to expand their knowledge in areas that are in support of their work. This is the case of the development of interoperability where the real problem to be addressed is limited to the interconnection of system interfaces and the data they exchange rather than at the intricacies of data serialization and communication protocols. Second, code generation can be optimal in order to adapt to the environment in which the generated code is executed. This allows replacing pieces of code that are too generic to be effective, too long to write to be optimal, or too difficult to maintain because too broad. Code generation is a potential candidate to enable better system interoperability.

Nonetheless, it is imperative that SMEs only see what concerns them in order to hide complexity from them, increase their productivity, and better leverage their expertise [48].

1.13 Model Configuration Management and Governance Deficiencies

Another problem with models created with the existing solutions is identifying changes and understanding how they evolved. This ability is critical in finding issues early on in the development and integration processes. This is also particularly significant in enabling model

governance which is an integral part of FACE's strategy to reduce software costs [26]. This also touches configuration management which is handled by revision control systems [58]. Contrast the differences between multiple revisions of a UML diagram or an XML file to that of a C++ or Java file under revision control. The latter source code representations enable simpler understanding of a model's evolution being clutter-free formats.

That is one of the motivations of [42] for migrating all of their UML models to DSLs. They also highlight the risk of UML models and source code getting out-of-sync which directly impacts the outcome of model governance activities: models are updated and source code needs to be in agreement with the changes. That is why they propose to store models as textual DSL representations alongside source code and applying the same software engineering practices, including reusing the same revision control system and configuration management. This is in line with [59] which proclaims that the model is the code in the context of service models. Treating DSL models as source code enables simpler understanding of a model's evolution. That is because a DSL provides a clutter-free view of the model and therefore allows for easier understanding, management, and governance.

1.14 Summary

This chapter presented an in-depth literature review covering system integration issues related to data exchanges, as well as issues impeding reuse and interoperability. As demonstrated, no single existing solution solves these problems on its own. Except each one offers a piece of the solution with some pieces intersecting while others diverge in their approach. Moreover, no solution provides an architecture-agnostic way of capturing system interfaces, i.e., meta-model, nor addresses multi-architecture considerations. The review also focused around two system integration perspectives, distributed simulations and platforms, generalizing both under the same roof.

Moreover, the review segmented the problem domain into the system interoperability facets. The findings highlight many issues related to the *Data* facet ranging from meta-model and data incompatibilities to how data models are captured. Another finding is the complexity incurred on SMEs when using existing solutions to model system data exchanges and the various aspects surrounding them. Furthermore, deficiencies of some solutions concerning model configuration management and governance point to potential areas of improvement and gains in efficiency.

The following chapter presents the research questions and objectives, as well as the motivation behind the proposed methodology.

Chapter 2 GENERAL METHODOLOGY

2.1 Research Motivation

Today’s platforms, such as full mission simulators (FMSs), exhibit an unprecedented level of hardware and software system integration. In this context, system integrators face heterogeneous system interfaces which need to be aligned and interconnected together in order to deliver a platform's intended capabilities. The sole aspect of the data systems exchange is problematic ranging from data misalignment up to multi-architecture environments over varying kinds of communication protocols. Similar challenges are also faced by integrators when interoperating multiple platforms together through distributed simulation environments where each platform can be seen as a system with its own distinct interface. On the other hand, enabling system reuse across multiple platforms for product line support is challenging for system suppliers, as they need to adapt system interfaces to heterogeneous platforms therefore facing similar challenges as integrators. Furthermore, the introduction of system interface changes in order to respond to late business needs, or unforeseen performance constraints for instance, is even more arduous as impacts are challenging to predict and their effect are often found late into the integration process. All these issues highlight the need to **simplify system integration and interoperability in order to reduce their associated costs and increase their effectiveness along with their efficiency**.

2.2 Problem Statement

Today's approach of achieving a high level of integration and interoperability between systems involves formal interface descriptions. Moreover, multi-architecture environments incur a duplication of these descriptions as each architecture requires its own representation. Introducing changes in this context becomes even more challenging as each representation might require a specific solution to address each change. From the perspective of system integrators, there is a need to have architecture-agnostic descriptions providing a single architectural viewpoint overseeing all system interfaces. On the other hand, suppliers require a method to adapt a system's interface to various platform-specific ones in order to enable reuse and better support product lines.

There exists a multitude of solutions to capture interface descriptions ranging from document specifications to dedicated languages. Some solutions enable the automation of some of the tasks involved in achieving system interoperability by generating software artifacts from interface descriptions. Some suggest using domain-specific languages (DSLs) to model the aspects relevant to specific stakeholders in the language of their respective domain therefore simplifying their comprehension.

Unfortunately, no single solution offers a single architectural viewpoint capturing all the details surrounding data exchanges covering system *Interface*s, the *Connection* of these interfaces to data, the *Data* exchanged between systems, and the data's *Transport* from system to system (Figure 2). They also exhibit limited validation capabilities. Moreover, interface descriptions captured with these solutions are not easily understandable even if some are expressed in human-readable formats because they do not target the system integrators or the SMEs involved in the development and integration activities. In turn, identifying changes and understanding how they evolved becomes challenging. This ability is critical in finding issues early on in the development and integration processes. This is also particularly significant in enabling interface governance. Furthermore, no solution addresses multi-architecture environments nor provides the flexibility required to enable system reuse across multiple platforms in support of product lines.

Therefore, there is a lack of an architecture-agnostic format which captures the details relevant to system data exchanges down to specific architectures, exhibits flexibility over system interfaces to enable their reuse, and is at the same time human-understandable by its stakeholders and machine-processable being exempt from ambiguity in addition to misinterpretation.

Consequently, this thesis tackles the general question of: **How should system interface descriptions be captured, and used, to simplify system integration and interoperability?**

2.3 Research Questions

The specific research questions presented in the following sections are meant to bring new advances in the fields of system integration and system interoperability. Notably, by establishing a common taxonomy, and by increasing the understanding of system interfaces, the various aspects impacting system data exchanges, multi-architecture environment considerations, and the factors enabling interface governance as well as system reuse.

2.3.1 What should be Formally Described in Order to Capture System Interfaces and the Various Aspects Surrounding their Data Exchanges, and How?

There is a lack of a common taxonomy which can be shared amongst stakeholders, such as integrators, suppliers, and system experts, when discussing system integration and interoperability. This taxonomy would allow a better understanding of the elements impacting system integration and system interoperability. The system interoperability facets (Figure 2) propose a breakdown based on the specific concerns surrounding system interoperability. Unfortunately, no single solution covers all of the facets, along with the concepts required to formally describe system interfaces, on its own. The elements relevant to stakeholders have been identified in the literature review (Chapter 1) and provide a good starting point for answering this question.

Additionally, existing solutions each provide a specific way of capturing the elements relevant to system interface descriptions. A common system interface description interchange format would enable better communication between stakeholders, simplify interface governance, enable reuse of system interface descriptions, and provide common grounds for engineering tools.

2.3.2 How should Multi-Architecture Considerations be Captured?

No existing solution addresses multi-architecture considerations which is a key problematic area of system integration and system interoperability. Identifying a way to capture such detail with the right level of abstraction would greatly benefit the integration and interoperability activities as well as further enable their automation. This would also enable easier change propagation between each architecture's representation of system interface descriptions.

2.3.3 How should System Interface Descriptions be Used to Automate Some of the Tasks Involved in System Integration and Interoperability?

The tasks involved in achieving system interoperability, such as data serialization or protocol handling, need to be formalized to remove ambiguity or misinterpretation. Some solutions derive software artifacts from interface descriptions in order to automate these tasks. Understanding how

to further enable the automation of the system integration and interoperability tasks would reduce costs and improve effectiveness, as well as efficiency, in performing them.

2.4 Research Objectives

The fundamental research objective of this thesis is to **automate the system-level data-interchange software through a system interface description language**. To this end, the following specific objectives have been derived.

2.4.1 Define a System Interface Description Language

This objective aims at defining a language used to describe system interfaces and the various details surrounding their data exchanges. This language is denoted as the *System Interface Description Language* (SIDL).

First, meeting this objective involves answering the research question "What should be Formally Described in Order to Capture System Interfaces and the Various Aspects Surrounding their Data Exchanges, and How?" (refer to Section 2.3.1). To this end, all the elements relevant to the system interoperability facets (Figure 2) are developed in Chapter 3 as the core language elements of SIDL. Moreover, [42] suggests using DSLs to model and capture the aspects relevant to specific stakeholders in the language of their respective domain. They also suggest storing models as textual DSL representations alongside source code to leverage the same software engineering practices, including reusing the same revision control system and configuration management. Therefore, the developed language is a textual DSL which is populated from elements of existing languages and elements derived from the author as well as SME feedback. Being a DSL, the proposed language is at the same time human-understandable, since in the language of its stakeholders, and machine-processable thus enabling automation. Therefore, this simplifies change identification and the understanding of interface evolution. Furthermore, the use of a model compiler enables strong validation semantics on the system interface descriptions.

Second, this objective also implies addressing the research question "How should Multi-Architecture Considerations be Captured?" (refer to Section 2.3.2). The language elaborated in Chapter 3 is architecture-agnostic such that architecture-specific representations can be derived from this single viewpoint. Because SIDL provides a single architectural viewpoint overseeing all system interfaces, SIDL is considered an architecture description language [3]. Moreover, in

order to address multi-architecture environment considerations, the language models architecture-specific details. Therefore, the introduction of changes to system interfaces becomes simplified as changes are automatically propagated down each architecture's own representation being introduced from a single architectural viewpoint defined in SIDL.

2.4.2 Define a Method to Automate the System-Level Data-Interchange Software from System Interface Descriptions

Achieving this objective directly answers the research question "How should System Interface Descriptions be Used to Automate Some of the Tasks Involved in System Integration and Interoperability?" (refer to Section 2.3.3). One way of simplifying system integration and interoperability is by automating some of its tasks. Therefore, this objective targets the automation of the software responsible for system data exchanges from system interface descriptions.

The general methodology of transforming system interface descriptions into data-interchange software will be elaborated in Section 3.6. The generated software artifacts that emanate from this process are largely contextual since they depend on characteristics such as the resulting software interface presented to the system, the frameworks or libraries which can be used in support of code generation, the output programming language, etc. This explains the generic trait of the elaborated method. That is why its experimental implementation is detailed in Chapter 4 which can form the baseline for other implementations.

2.5 General Approach

The System Interface Description Language (SIDL) has been developed along with a method for automating the data-interface software of systems from descriptions expressed in the elaborated language. These two results demonstrate the achievement of the specific research objectives which leads to a better understanding of system integration and interoperability as well as enables the simplification of their associated activities.

Because of the intrinsic relationships between each research objective, they have all been approached at the same time with the following iterative methodology:

1. Identify use cases
2. Identify and implement test cases, composed of test systems, covering all use cases
3. Define language syntax & semantics
4. Prototype language implementation
    4.1. Model test system interfaces and data exchanges in language
    4.2. Generate software artifacts realizing data-interchange for each test system
    4.3. Refactor test systems
    4.4. Validate results
    4.5. Improve language
    4.6. Go back to 4.1 if all use cases are not achieved or if data-interchange software requires manual intervention, otherwise stop.

The methodology starts with the identification of use cases which are populated mainly from the author's experience with support from SME feedback and literature data (detailed in Chapter 1 in the context of full mission simulators). This enables use cases to be more pragmatic covering existing system integration and interoperability concerns. Then, a series of test cases, composed of multiple test systems, are elaborated in order to cover all use cases. Each test case can verify multiple use cases, and use cases can be verified by more than one test case. This overlap is intentional and allows for greater certainty over the expected results. At this stage, the data-interchange software of each test case is manually created with the intent of being automated at a later stage. A language baseline is created next based on literature data which particularly focuses on standard languages.

From there on, an iterative approach starts for converging both the language and the data-interchange software automation until all use cases are demonstrated, and all test cases no longer require manual intervention to use the generated software interface. Each iteration sees more and more pieces of the test systems become automated now being modeled with the language. In turn, this has the effect of modifying the software interface consumed by the test systems which justifies the need for a refactoring activity to occur. Once the test systems compile, their behavior is validated to prevent regressions from the expected test results. At this stage, SMEs also

validate the language constructs with the primary end goal being to improve its human-understandability. These results are then injected into the next and final stage which consists of improving the language based on them.

This general overview of the iterative methodology is revisited in detail in Chapter 4 particularly when describing the experimental implementation used to meet the research objectives. The achievement of each research objective is demonstrated in the following sections.

2.5.1 System Interface Description Language

The demonstration of this objective revolves around three aspects: the relevant language elements, the capture of models described with the language, and addressing the multi-architecture considerations.

The first aspect revolves around the language elements, i.e., *What* should formally be described by the language. This is demonstrated as SIDL covers all the system interoperability facets (Figure 2) as depicted in its conceptual model (Figure 3-1). Each language element is categorized under one of the facets notably the system *Interface*s, the *Connection* of these interfaces to data, the *Data* exchanged between systems, and the data's *Transport* from system to system. As such, each element targets a specific aspect of system integration and interoperability as described throughout Chapter 3. This is also demonstrated by the fact that all SIDL elements are derived from standard language elements dealing with integration and interoperability, notably FACE [26], WSDL [28], AADL [40], and HLA OMT [17], as well as from the author and SME feedback.

The second aspect concerns *How* to capture the language. This is demonstrated since SIDL is represented as a textual DSL, as such is human-understandable, being in the language of its stakeholders, and machine-processable being equally considered as source code. Additionally, being supported by a model compiler (Chapter 4), SIDL demonstrates strong validation semantics on system interface descriptions as detailed in Section 3.7. The success of its validation semantics has also been praised by SMEs using it as illustrated in Chapter 5.

The third aspect concerns addressing multi-architecture considerations. This is demonstrated as SIDL is architecture-agnostic and can model architecture-specific details through its *Transport* elements as detailed in Section 3.5. Moreover, architecture-specific representations can be

derived from SIDL descriptions as demonstrated in the experimental results detailed in Chapter 5 when generating IDL [32] and HLA OMT [17] data exchange model representations.

2.5.2 System-Level Data-Interchange Software Automation

The demonstration of this objective revolves around the automation of the data-interchange software used by systems. This is demonstrated by having the data-interchange software of all the test systems be entirely generated from models described in SIDL. Each test system is represented in SIDL covering each system interoperability facet, i.e., the test system *Interface*s, the *Connection* of test systems to data, the *Data* the test systems exchange, and the *Transport* of data between test systems. From these definitions, a code generator creates the corresponding software artifacts which each test system uses. This process is detailed in Chapter 4 in the context of the experimental implementation used to demonstrate the research objectives. The generated software artifacts include:

- the software interface expressed in one of the supported programming languages covering a representation of the data exchange model and a communication interface supported by frameworks and libraries
- the data serialization software which is expressed in either one of the supported programming languages or in a dedicated language such as IDL in the case of DDS
- the communication middleware data exchange model representations such as HLA OMT data exchange models.

The software artifacts that emanate from this process are specific to the experimental results hence the objective is only demonstrated in this context. In order to extrapolate and generalize the demonstration to other contexts, a general methodology of transforming system interface descriptions into data-interchange software artifacts is elaborated in Section 3.6. Moreover, the resulting data-interchange software is validated through regression tests as well as objective criteria instead of empirical measurements. The hypothesis is that the generated artifacts are equivalent, or better, to the ones an expert would have produced. This is validated as the code generation process uses code templates produced by experts.

2.5.3 Publications

During the course of this work, five conference articles have been published. Each article focuses on specific aspects related to the research objectives.

- Conference Article #1 ([60]): Study on using code generation from simulation data exchange model representations and interoperability mapping descriptions to automate the data-interchange software of gateways for system interoperability. [SISO Fall SIW 2005, 05F-SIW-074]
- Conference Article #2 ([61]): Study on using XML formats for describing simulation data exchange models and system interoperability mappings by SMEs to automate the data-interchange software of gateways dynamically through runtime code generation. [SISO Spring SIW 2006, 06S-SIW-087]
- Conference Article #3 ([62]): Study on performance of gateways created using XML interoperability descriptions through runtime code generation. [SISO Spring SIW 2006, 06S-SIW-086]
- Conference Article #4 ([48]): Study on hiding software complexity from SMEs in order to better leverage their expertise. [I/ITSEC 2011, 11236]
- Conference Article #5 ([63]): Study on a system interface description language for simplifying system integration and interoperability. [SISO Fall SIW 2013, 13F-SIW-021]

Part II

METHODOLOGY AND RESULTS

Chapter 3 SYSTEM INTERFACE DESCRIPTION LANGUAGE

SIDL is a language used to formally describe system interfaces focusing on the data they exchange and on the various aspects surrounding them. This information set is captured into one or more SIDL *descriptions*. Figure 2 illustrates the facets covered by SIDL descriptions notably the system *Interfaces*, the *Connection* of these interfaces to data, the *Data* exchanged between systems, and the data's *Transport* from system to system.

The conceptual model of SIDL is illustrated in Figure 3-1. It presents the high-level elements of a SIDL description and their relationships to one another. The elements are also categorized against the facet they relate to and are detailed alongside this organization throughout the following sections. It is important to note that the *Interface*, *Connection*, and *Data* facets are considered conceptual whereas *Transport* is concrete. This implies that any conceptual element is designed to be reusable by other SIDL descriptions. Concrete elements can be reused, but with less extent.

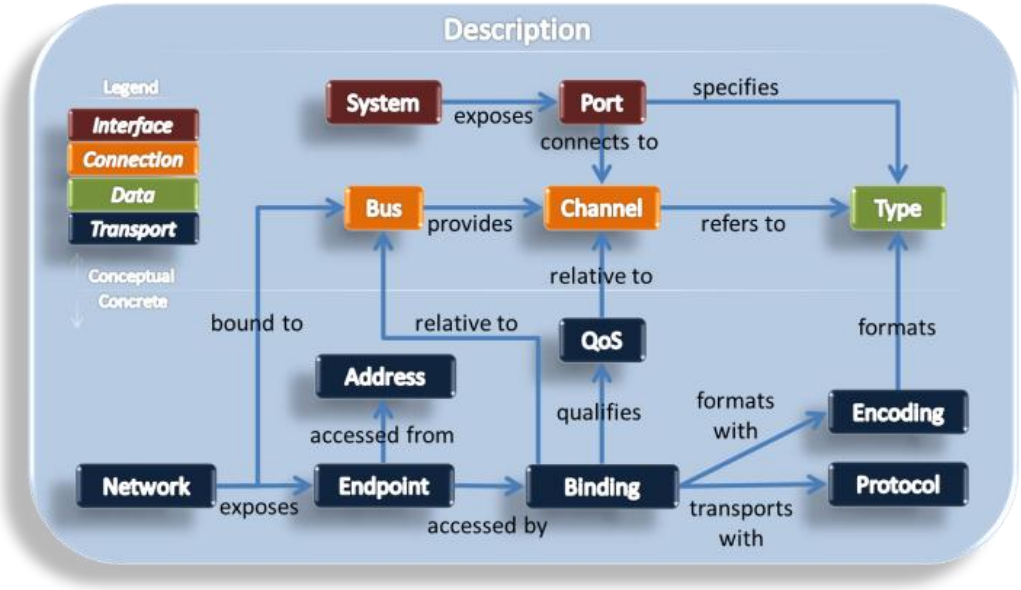

Figure 3-1: SIDL Conceptual Model

SIDL's language elements are primarily derived from FACE [26], WSDL [28], AADL [40], and HLA OMT [17], while some originate from the author as well as SME feedback. Throughout the following sections, each element identifies its origin to better position its relationship to system integration and interoperability.

Note that the examples used throughout this section are inspired by RPR-FOM [64]. Moreover, some examples refer to declarations of previous ones.

3.1 SIDL Grammar

SIDL aims to be a human-understandable language. As such, it reads very similarly to the English language with the exception of being structured differently. Another difference with typical programming languages is that SIDL uses English words instead of relying on symbolic operators. For instance, to denote the type of a field, the **as** construct is used instead of typical whitespace or a colon ':' delimiter. The following example illustrates these differences.

```
// SIDL
Psi as AngleRadian

// C/C++/Java/C#
AngleRadian Psi;

-- ASN.1
Psi AngleRadian

-- AADL
Psi: data AngleRadian;
```

The preceding example also demonstrates the different single-line comment styles. In addition, multi-line comment are enclosed between '/*' and '*/' in SIDL as with C, C++, Java, C#, and ASN.1. Note that AADL does not support multi-line comments.

3.1.1 Control Blocks

Another difference with SIDL is the lack of explicit control block delimiters. SIDL uses whitespace instead whereas most languages use delimiters such as curly braces '{}', or keywords such as '**end**' paired with a start keyword. One limitation of this approach is that multiple declarations cannot be provided on the same line. This limitation is overcome with the improved readability this incurs on SIDL descriptions. The following example illustrates these differences.

```
// SIDL
entity Orientation:
  Psi as AngleRadian
  Theta as AngleRadian
  Phi as AngleRadian

// C/C++
struct Orientation
{
  AngleRadian Psi;
  AngleRadian Theta;
  AngleRadian Phi;
};
```

```
// C#
struct Orientation
{
  AngleRadian Psi;
  AngleRadian Theta;
  AngleRadian Phi;
}

// Java
class Orientation
{
  AngleRadian Psi;
  AngleRadian Theta;
  AngleRadian Phi;
}

-- ASN.1
Orientation ::= SET
{
  Psi AngleRadian,
  Theta AngleRadian,
  Phi AngleRadian
}

-- AADL
data Orientation
subcomponents
Psi: data AngleRadian;
Theta: data AngleRadian;
Phi: data AngleRadian;
end Orientation;
```

3.1.2 Namespaces and Imports

In order to better structure SIDL descriptions, namespaces can be used. A namespace is declared using the `namespace` SIDL element, and must appear as the first declaration in a SIDL description. The following example demonstrates namespaces in SIDL.

```
namespace Rpr

entity OrientationStruct:
  Psi as AngleRadian
  Theta as AngleRadian
  Phi as AngleRadian
```

The preceding example attributes the `Rpr` namespace to the `Orientation` entity. Therefore, referencing the orientation entity now requires its fully qualified name which becomes `RprFom.Orientation`. In order to simplify referencing SIDL declarations, import declarations can be used. An import is declared using the `import` SIDL element. This is demonstrated in the following example.

```
import Rpr

entity SpatialStaticStruct:
  Orientation as OrientationStruct
```

3.1.3 SIDL Source File Encoding

All SIDL source files must conform to the UTF-8 character encoding as specified by ISO/IEC 10646-1 [65].

These are the main elements of the SIDL grammar and are used throughout the following sections when providing SIDL examples. Appendix A provides the full SIDL grammar reference.

3.2 The Data Facet

The data available to be exchanged between systems is the concern of the *Data* facet. It takes form as the SIDL *Data Model* which is illustrated in Figure 3-2.

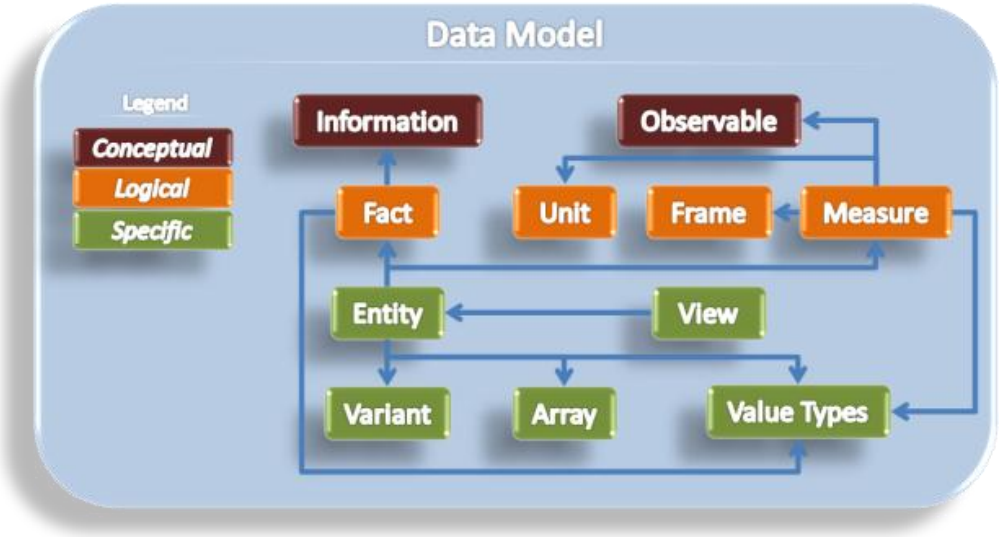

Figure 3-2: SIDL Data Model

The SIDL data model is derived from the FACE data model as its main purpose is "to provide an interoperable means of data exchange" [26]. The FACE data model also provides many elements which increase the semantic meaning of data such as units of measurement and frames of reference. Moreover, FACE promotes reuse extensively which its data model reflects by enabling it. For all these reasons, the SIDL data model can be seen as a DSL representation of the FACE data model. However, some divergences are introduced notably: the addition of facts (Section 3.2.2.1), variations of the supported value types (Section 3.2.3.3), the support for variants (Section 3.2.3.5), the support for mapping existing data models (Section 3.2.4), and the concept of a concrete reference data model (Section 3.2.5).

The SIDL data model is subdivided into three levels of abstraction being, from highest to lowest, *Conceptual*, *Logical,* and *Specific*. The following sections describe each level in detail.

3.2.1 Conceptual Data Model

The Conceptual Data Model (CDM) captures the fundamental elements of *Data* which are informations and observables. The objective of the CDM is to provide the highest level of abstraction to data such that it exhibits the highest level of reuse. The CDM is meant to be refined by the logical data model such that it increases the semantic meaning of the CDM elements. Therefore, the CDM elements create the unifying link between all of their refined representations.

3.2.1.1 Informations and Observables

An information is "something that is typically not quantified through measurement of the physical world but is descriptive in nature" [26]. It is declared using the `info` element followed by its name. Name, description, and unique identity are information examples. The following SIDL description presents information examples.

```
info Name
info Description
info UniqueIdentity
info PartialIdentity
```

An observable is "something which can be quantified through measurement(s) of the physical world" [26]. It is declared using the `observable` element followed by its name. Speed, pressure, and mass are examples of observables. Examples of observables expressed in SIDL follow.

```
observable Angle
observable Orientation
```

Together, informations and observables allow strong semantic links to be established with any of their logical refinements. For instance, the `Angle` observable can be used to relate to any angle representation independently from its computing platform representation, units of measurement, or frame of reference.

3.2.2 Logical Data Model

The Logical Data Model (LDM) extends the CDM through refinement. The objective of this refinement is to increase the semantic meaning of CDM elements while offering concrete data representations of the them to suit specific needs. Notably, the logical elements capture facts and

measures which provide the concrete representations of informations and observables respectively augmented with semantic information such as engineering units and frames of reference.

3.2.2.1 Facts

A fact is one or more evidences representing an information. It is not part of the FACE data model and has been introduced by the author. Its purpose is to provide a concrete representation for an information which FACE does not capture. It is declared using the **fact** element followed by its name. The **of** construct is then used to establish the link between the fact and the information it concretely refines by specifying the information reference after the construct. Before delving into the purpose of this link, which is detailed in Section 3.2.2.3, a closer look at facts is required. Facts are subdivided into two categories: simple and composite.

A simple fact reduces an information to a single evidence. The fact's representation is specified using the **as** construct followed by the representation's reference which specifies its concrete value type (Section 3.2.3.3). Follows is the concrete representation of a partial identity as a 16-bit unsigned integer fact.

```
// Simple fact refining the PartialIdentity info
fact IdPart of PartialIdentity as ushort
```

A composite fact reduces a set of informations to a set of evidences. It is composed of one or more facts, simple or composite. An entity identifier fact representing a unique identity information is illustrated next.

```
// Composite fact composed of simple facts and refining the UniqueIdentity info
fact EntityIdentifier of UniqueIdentity:
  Site as IdPart
  AppId as IdPart
  EntId as IdPart
```

3.2.2.2 Units, Frames, and Measures

A measure is “one or more quantities representing an observable in a defined frame of reference” [26]. It is declared using the **measure** element followed by its name. The **of** construct is then used to establish the link between the measure and the observable it concretely refines by specifying the observable reference after the construct. As measures are the parallel of facts for observables, this link, which is detailed in Section 3.2.2.3, enables an observable to relate to all of its

corresponding measure representations. Additionally, measures are subdivided into two categories: simple and composite.

A simple measure "reduces an observable to a single quantity that can be recorded" [26]. It specifies this quantity through its units, frame of reference, and precision. Units of measurement are declared using the `unit` element followed by its name. Associating a unit to a simple measure is done by referencing the unit using the `units` property on the measure. The same applies to frames of reference which are declared using the `frame` element followed by its name, and are associated to a measure using the `frame` property. The precision of the measure's value is expressed with the `precision` property as a real literal value (i.e., decimal value). The measure's representation is specified after the `as` construct which specifies its concrete value type (Section 3.2.3.3). The following example illustrates an angle measure in radians with a True-North reference frame.

```
unit Radian
frame TrueNorth

// Simple measure refining the Angle observable
measure AngleRadian of Angle as single:
  units Radian
  frame TrueNorth
  precision 0.000001
```

A composite measure "reduces a set of observables to a set of quantities that can be recorded" [26]. A composite measure can be composed of other measures, simple or composite. It can specify a frame of reference which its composed measures become relative to. The following example illustrates the concrete representation of an orientation observable as a composition of angle measures.

```
// Composite measure composed of simple measures refining the Orientation observable
measure OrientationMeasure of Orientation:
  Psi as AngleRadian
  Theta as AngleRadian
  Phi as AngleRadian
```

3.2.2.3 Linking Conceptual and Logical Elements

The purpose of linking conceptual and logical elements together is to increase the semantic meaning of data. As an information/observable is refined concretely by one or more fact/measure through the `of` construct, a new semantic link is created that is shared between the fact/measure with the notion that they all relate to the same information/observable. This is very powerful as many data exchange models represent the same kind of data except with different computing

platform representations or semantics. This particularly impacts heterogeneous system integration. In the context of multi-architecture environments, it becomes even more essential to establish and preserve such links as each architecture has its own representation of a data model. Many interoperability issues originate from the lack of such links.

As an example, consider the concept of *unique identity* which is present in many data models. Within the RPR-FOM [64], for instance, there exists many kinds of unique identities notably: entity identity (`EntityIdentifier`), object identity (`RTIobjectId`), and emitter beam identity (`BeamIdentifier`). All have different computing platform representations with varying size and structure. Additionally, no link exists which relates any of them together. Interoperating RPR-FOM data with other data exchange models adds even more kinds of unrelated unique identities. This prevents from automatically treating identities with a common pattern. Increasing the semantic level of data through informations and facts would add the missing links, and could enable better automation particularly in the context of gateway applications.

Another example is in regards to the semantic meaning of data. One common problem faced in interoperating data exchange models is with the concept of spatial positions. The typical geodetic versus geocentric position is always a problematic area because both are often represented with the exact same computing platform representation, except their semantic meaning are completely different. Likewise, consider a UUID, which is a universally unique identifier represented as a 128-bit value, and four 32-bit integers. Both types exhibit the same size, i.e., 128-bit, except their semantic meaning are not aligned. Again, increasing the semantic meaning of data by establishing links between conceptual and logical elements would prevent these problems.

3.2.3 Specific Data Model

The Specific Data Model (SDM) provides concrete data representations. Its main objective is to compose logical elements together and expose them as higher-level concepts. Enabling system reuse is another SDM objective, and is realized through views which enable the adaption of system interfaces to *Data* (refer to Section 3.2.3.6). Another objective of the SDM is to enable the representation of existing data models in SIDL by providing concrete representations aligned with them (detailed in Section 3.2.4). The principal SDM elements are entities, value types, and views.

3.2.3.1 Entities

An entity is a “non-basis concept that is constructed through composition of basis elements and other entities” [26]. An entity can be seen as a structure, or fixed record, with fields. Each field has a name, and is of a specific type which can either be another entity or a basis element. Basis elements are either facts, measures, or value types. Moreover, an entity is declared using the **entity** element followed by its name. Fields are constructed by specifying their name followed by the type reference using the **as** construct. The following example demonstrates this.

```
entity WorldLocationStruct:
  // Value type composition
  X as double
  Y as double
  Z as double

entity BeamAntennaStruct:
  // Composite measure composition
  Orientation as OrientationMeasure

  // Simple measure composition
  AzimuthWidth as AngleRadian
  ElevationWidth as AngleRadian
```

A cardinality can be specified on a field's type to indicate a bounded or unbounded array, i.e., a collection of items, using the array construct "`(type)`". The following SIDL example demonstrates arrays in SIDL.

```
entity SphericalAntennaStruct:
  // Unbounded array composition
  OrderACoefficients as (single)

entity MarkingArray11:
  // Bounded array composition
  Items as (byte, 11)
```

3.2.3.2 Composition Over Polymorphism

As with FACE, SIDL favors composition over polymorphism, i.e., inheritance. This constraint ensures coherence in views (Section 3.2.3.6) because views navigate the fields of entities. Supporting polymorphism on entities would create complexity in specifying the navigation paths particularly when inherited types expose fields of the same name as in derived ones. This phenomenon, called name hiding, would require SMEs to disambiguate to correct field to select by having them qualify the field's type. For this reason, SIDL prevents polymorphism. Moreover, variants can be used to emulate it (refer to Section 3.2.3.5).

3.2.3.3 Value Types

A value type specifies a specific data representation aligned with computing platforms. Most value types have a fixed size in bytes. They are derived from a combination of FACE value types [26], .NET primitive types [66], and Boo value types [67]. Basically, the value types are aligned with FACE with the exception of `decimal`, which comes from .NET and provides a greater numerical range than `long double`, and `string` as well as `char` which use UTF-8 for Unicode character support [65]. The type names originate from Boo to make them more succinct, and are very similar to C# [66] with the exception of `single` which replaces `float` providing a more natural alignment with `double`. Table 3-1 presents the SIDL value types.

Table 3-1 SIDL Value Types

Value Type	Range	Size (bytes)
`sbyte`	-128 to 127	1
`byte`	0 to 255	1
`short`	-2^{15} to $(2^{15} - 1)$	2
`ushort`	0 to $(2^{16} - 1)$	2
`int`	-2^{31} to $(2^{31} - 1)$	4
`uint`	0 to $(2^{32} - 1)$	4
`long`	-2^{63} to $(2^{63} - 1)$	8
`ulong`	0 to $(2^{64} - 1)$	8
`single`[1]	~ $\pm 1.5e^{-45}$ to $\pm 3.4e^{38}$	4
`double`[1]	~ $\pm 5.0e^{-324}$ to $\pm 1.7e^{308}$	8
`decimal`[2]	~ $\pm 1.0e^{-28}$ to $\pm 7.9e^{28}$	16
`bool`	true or false	1
`char`	Unicode character (UTF-8)	1
`string`	Unicode string (UTF-8)	Unbounded `char` sequence
`enum`	2^{32} identifiers	4

[1] IEEE floating-point numbers (single and double)

[2] .NET System.Decimal number

3.2.3.4 Enumerations

An enumeration is "an ordered list of identifiers" [26]. As opposed to the other value types, enumerations are not predefined in SIDL hence its different color coding which distinguishes it from the predefined ones. An enumeration is declared using the `enum` element followed by its name. An enumeration is a collection of enumerators which are specified using a name. Optionally, an integer literal value can be associated to an enumerator. The following example demonstrates enumerations in SIDL.

```
enum AntennaPatternEnum:
  Beam
  SphericalHarmonic

enum DeadReckoningAlgorithmEnum8:
  Other = 0
  Static = 1
  DRM_FPW = 2
  DRM_RPW = 3
  DRM_RVW = 4
  DRM_FVW = 5
  DRM_FPB = 6
  DRM_RPB = 7
  DRM_RVB = 8
  DRM_FVB = 9
```

3.2.3.5 Variants

A variant is a special kind of entity. Variants are "discriminated unions of types" [17]. They are present in many languages such as C, C++, IDL, and HLA OMT. A variant is used to represent a value which has a finite set of varying forms such as for a uniform array of different items. The end result is similar to polymorphism (i.e., inheritance). That is primarily why HLA OMT incorporates such a concept. A variant's fields are called alternatives, and must conform to entity fields (refer to Section 3.2.3.1). The alternative to use is determined by the variant's discriminant value which is usually an enumeration.

A variant is declared using the `variant` element followed by its name. Then follows the reference to the discriminant's type which is specified using the `of` construct. The list of alternatives is specified using `case` constructs based on the discriminant's range of values. The value associated to an alternative must not already be specified by another one. A default alternative can be specified using the `otherwise` construct, and must appear after all declared alternatives. A variant must specify at least a single `case` or `otherwise`. The following example demonstrates how to declare a variant in SIDL.

```
variant AntennaPatternType of AntennaPatternEnum:
  // BeamAntenna alternative accessible only when discriminant's value is Beam
  case AntennaPatternEnum.Beam:
    BeamAntenna as BeamAntennaType

  // Otherwise, the SphericalAntenna alternative is accessible
  otherwise:
    SphericalAntenna as SphericalAntennaType
```

3.2.3.6 Views

A view is a particular way of representing one or more entities. The concept of views originates from FACE [26]. A view can be seen as a window on entities. It can be used to specify particular interest in a subset of an entity's fields. It can also be used to adapt data such as units, frames, and representations. Even names can be adapted with a view which provides an aliasing mechanism in case where a different notation is required. All these traits make views the pattern of choice when requiring *Data* adaptation.

A view is declared using the **`view`** element followed by its name. Specific points of interest are specified through **`select`** constructs. This allows navigation within the composed entity's fields through the dot '.' path delimiter. The end result of selecting an entity's fields is that they become an integral part of the view. The selected field's name can be changed by specifying the new name with the **`alias`** property on the **`select`** construct. Views are particularly well suited to adapt entities as is demonstrated in the following example.

```
view AppAndWideEntityNumber:
  // Only interested in EntityIdentifier's AppId and EntId, i.e., ignore Site
  select EntityIdentifier.AppId
  // Adapt EntId's representation from ushort to uint
  select EntityIdentifier.EntId as uint:
    // Adapt name using an alias
    // i.e. EntityNumberWide contains an EntityNumber field
    //      instead of EntId as in EntityIdentifier
    alias EntityNumber
```

In the previous example, the `AppAndWideEntityNumber` view shows an interest in some of `EntityIdentifier`'s fields by only selecting a subset of them. The end result is similar to having defined a `AppAndWideEntityNumber` entity with `AppId` and `EntId` fields. Moreover, the `EntId` field is adapted from **`ushort`**, i.e., `EntityIdentifier`'s representation, to **`uint`** using the **`as`** construct. The field's name is also changed by renaming it to `EntityNumber` to adapt to a different notation. Therefore, `AppAndWideEntityNumber` is equivalent to the following declaration.

```
entity AppAndWideEntityNumber:
  AppId as ushort
  EntityNumber as uint
```

The `BeamAntennaDegrees` view shows an interest in `BeamAntennaStruct`'s fields by selecting all its fields. The end result is similar to having defined a `BeamAntennaDegrees` entity with `AzimuthWidth` and `ElevationWidth` fields. The following example demonstrates unit adaptation where `BeamAntennaDegrees` requires angle specified in degrees instead of radians.

```
// BeamAntennaStruct uses fields which are angles in radian
entity BeamAntennaStruct:
  AzimuthWidth as AngleRadian
  ElevationWidth as AngleRadian

unit Degree
measure AngleDegree of Angle as single:
  units Degree

// BeamAntennaDegrees uses fields which are angles in degrees
view BeamAntennaDegrees:
  // Adapt an angle in radians to an angle in degrees
  select BeamAntennaStruct.AzimuthWidth as AngleDegree
  select BeamAntennaStruct.ElevationWidth as AngleDegree
```

Views play an invaluable role in enabling system interoperability as is further discussed in Section 3.2.5.

3.2.4 Existing Data Model Support

SIDL enables value types to be specified directly on elements, such as entities, simple facts, and simple measures, which FACE disallows [26]. This support is mandatory in order to map existing data models. FACE forces the elaboration of strong semantics which is great for new development. Unfortunately, it is far from being trivial to map existing data exchange models to observables, informations, and measures for instance. Refactoring existing data models could also compromise the integrity of already deployed applications. Interoperating with such applications inevitably requires to represent them as they are. Another barrier to existing data model support is with enumeration literals which are disallowed to specify a literal value by FACE. This is primarily because of the IDL baggage which prevents this [32]. Except they are required by some data exchange models and therefore are captured in SIDL.

3.2.5 Concrete Reference Data Model

The main difference with the FACE data model is that SIDL elements belong to a single level of abstraction instead of having refined representations over multiple ones. For instance, an *entity* only has a *Specific* SIDL representation for it whereas FACE has *Conceptual*, *Logical,* and *Specific* ones. FACE's design decision allows for vaster data variability scenarios. Combined to

the fact that FACE governance is only applied at the conceptual and logical levels, this has the side effect of specific data possibly varying on every FACE platform causing divergences. These divergences might be required in order to deliver the specific platform, except they should be minimized. Therefore, constraining refinement in SIDL forces the emergence of a concrete reference data model because it inherently promotes specific data reuse. It also has the side effect of simplifying modeling by minimizing the repetitive refinement tasks found in FACE.

An analogy to this would be that SIDL promotes concrete representations whereas FACE promotes conceptual ones. Both FACE and SIDL promote conceptual and logical data reuse. Except over time, system integration can become simplified by having more and more systems adhering to a concrete reference data model dealing with variability only within the systems not doing so such as with views to support specific platform needs.

As an example, consider a concrete reference data model which suggests using angles in radians and expressed using `double` value representations. Derived data models can opt to not use this definition, except using it enables easier system reuse across platforms.

That is why some logical elements specify a value type as their reference representation in SIDL, and why some elements are considered specific whereas they have a higher-level of abstraction in FACE.

3.3 The Interface Facet

The *Interface* facet is used to capture system interfaces. System interfaces express the data systems require to consume, and the data they produce. This can be seen as a data-centric system contract and is captured in SIDL using system elements.

3.3.1 Systems and Ports

The term *system* is used to refer to entities whose interface is of interest. Moreover, the system element is the parallel of the interface element of WSDL [28] which is used to describe a service interface. The word system is used in SIDL instead of interface because it is systems who need to interoperate, not their interfaces which only need to be aligned.

A system is declared using the `system` element followed by its name. It is considered a black-box being solely composed of ports which make up its interface. Ports are similar to fields on entities

with the main difference being that a port can either **input** or **output** data, or perform both, i.e., **inout**. Following the port's flow is the port's name. The type of the port represents the data that flows through it in the form of messages. Therefore, a port restricts data to a single kind of message. The message type is specified using the **of** construct followed by the reference to the message type. The **of** construct distinguishes the port's messaging nature from the **as** construct used on fields which denotes their representation. Messages can be seen as instances of types with specific values.

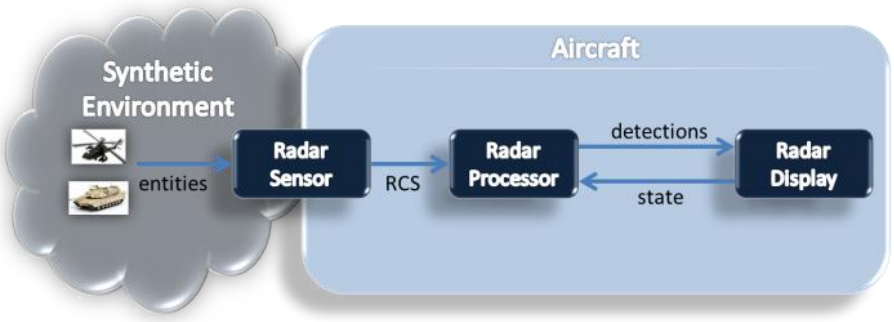

Figure 3-3: Radar System Example

The following example demonstrates systems in detail and is illustrated in Figure 3-3. The figure shows a simplified radar system. The aircraft's radar sensor extracts radar cross-section (RCS) information from entities it acquires within the synthetic environment. The RCS information is then passed to the processor which determines the ones it detects. The detection information is finally pushed to the display. The radar display also controls the state of the processor by turning it on or off. The SIDL description corresponding to the whole example is presented next.

```
// Entity description which originates from synthetic env. and is captured by the sensor
entity Entity:
  // Entity's unique identity
  EntityIdentifier as EntityIdentifier
  // RCS signature as DB index
  RcsSignatureIndex as short
  // Acoustic signature as DB index
  AcousticSignatureIndex as short

// Represents the RCS of an entity
entity RadarCrossSection:
  // Entity's unique identity
  EntityIdentifier as EntityIdentifier
  // RCS signature as DB index
  SignatureIndex as short
```

```
// Represents a list of RCS exchanged between the sensor and the processor
entity RcsList:
  // Limit RCS count
  Items as (RadarCrossSection, 500)

// Represents a detection
entity Detection:
  // Detected entity's unique identity
  EntityIdentifier as EntityIdentifier

// Represents a list of detections exchanged between the processor and the display
entity DetectionList:
  // Limit detection count
  Items as (Detection, 20)

// Possible processor states
enum RadarStateEnum:
  Off
  On

// Represents a radar state
entity RadarState:
  State as RadarStateEnum

// Extracts RCS info from entities which are processable by the radar processor
system RadarSensor:
  // Inputs Entity messages
  input Entities of Entity
  // Outputs RCS lists messages
  output RadarCrossSections of RcsList

// Transforms RCSs into detections which can be displayed by the radar display
system RadarProcessor:
  // Inputs radar state messages
  input State of RadarState
  // Inputs RCS lists messages
  input RadarCrossSections of RcsList
  // Outputs detection lists messages
  output Detections of DetectionList

// Displays the radar detections and control the radar's state
system RadarDisplay:
  // Inputs detection lists messages
  input Detections of DetectionList
  // Outputs radar state messages
  output State of RadarState
```

The preceding example contains two types of declarations: data and systems. The data exchanged between the radar system's components is captured with entities and value types, such as the `RadarState` enumeration. The system interfaces of each radar component are captured with system elements. Notice the symmetry between the inputs and outputs of systems. From the preceding SIDL description which captures *Data* and system *Interface*s, we know that systems share data, because ports use the same types, except we do not know where data actually flows.

3.4 The Connection Facet

Whereas the *Data* facet is concerned about *what* data is available to systems and the *Interface* facet about *which* data systems consume or produce, the *Connection* facet is concerned about

where data is routed. Therefore, the *Connection* facet is used to capture how system interfaces conceptually connect to one another. This is achieved using bus and channel elements.

3.4.1 Buses and Channels

A bus is a collection of channels which facilitates the connections and communication of systems as well as enable their interaction. It is inspired by the same concept in AADL [68] with the difference that the SIDL bus is conceptual. The conceptual links between the *Connection*, *Data*, and *Transport* facets is illustrated in Figure 3-4.

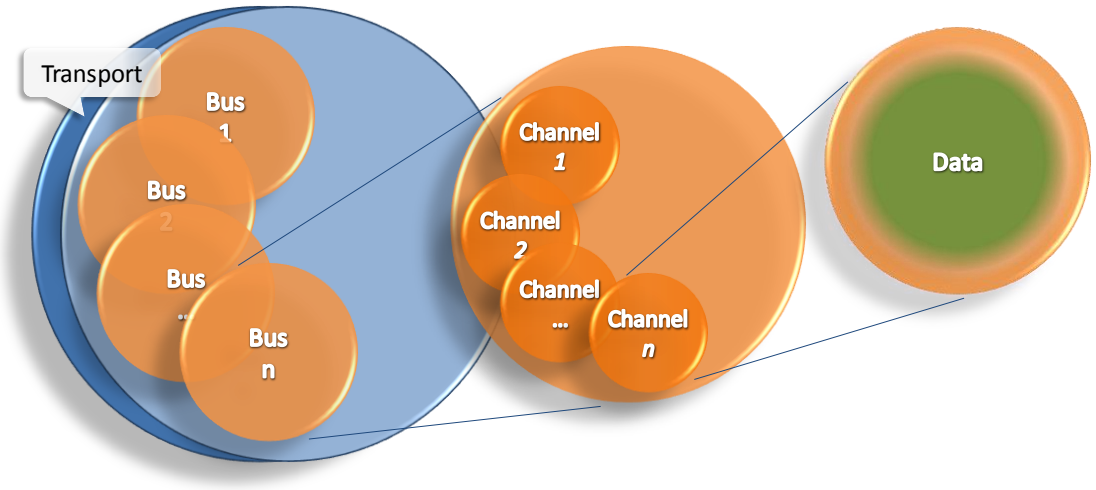

Figure 3-4: SIDL Conceptual Data Transport

A bus is declared using the **bus** element followed by its name. Channels are added to a bus using the **channel** construct followed by the channel's name. Each channel makes data available in the form of messages which is specified using the **of** construct followed by the reference to the message type. The bus for the radar system follows in SIDL.

```
bus RadarSystemBus:
  channel Entities of Entity
  channel RadarCrossSections of RcsList
  channel Detections of DetectionList
  channel RadarState of RadarState
```

The preceding example only captures the radar system's bus structure. In order to route data between systems, connection information needs to be captured. Systems connect their ports to channels using the **connect** construct on a channel followed by the port's reference. Therefore, the channel establishes the communication the link between all the ports connected to it. The full radar system's bus, including the system connection information, follows in SIDL.

```
bus RadarSystemBus:
  channel Entities of Entity:
    connect RadarSensor.Entities

  channel RadarCrossSections of RcsList:
    connect RadarSensor.RadarCrossSections
    connect RadarProcessor.RadarCrossSections

  channel Detections of DetectionList:
    connect RadarProcessor.Detections
    connect RadarDisplay.Detections

  channel RadarState of RadarState:
    connect RadarProcessor.State
    connect RadarDisplay.State
```

The `RadarSystemBus` now fully captures the system relationships found in Figure 3-3.

3.4.2 Configurable Routing

Multiple buses can be declared for the same systems with different connection information. This enables system instances to be configured with specific routing and is particularly useful when dealing with redundancy. For instance, an aircraft system is often paired with a pilot and co-pilot instance. Using separate pilot and co-pilot buses enables both instances to reuse the same system interface and be configured with different routings.

```
system Egi:
  output Attitudes of Attitude

system Imu:
  output Attitudes of Attitude

system Nav:
  input Attitudes of Attitude

bus LeftBus:
  // Use Egi's attitude
  channel Attitudes of Attitude:
    connect Nav.Attitudes
    connect Egi.Attitudes

bus RightBus:
  // Use Imu's attitude
  channel Attitudes of Attitude:
    connect Nav.Attitudes
    connect Imu.Attitudes
```

3.5 The Transport Facet

The previous facets have captured *what Data* is exchanged between systems, *which* data they consume as well as produce through their *Interface*, and *where* data is routed through the *Connection* of their interfaces to data. The missing link required to completely capture system

interoperability is *how* this data gets transported from system to system, and is the concern of the *Transport* facet.

The *Transport* facet establishes the concrete link required to address details such as protocol and encoding which explains why this facet is considered concrete whereas the others are considered conceptual (Figure 3-1). For this reason, these elements are primarily targeted towards system integrators. The transport elements include binding, protocol, network, and endpoint. Basically, a binding capture the protocol details of buses which a network makes accessible through endpoints.

In addition, the transport elements are derived directly from WSDL [28]. The only divergence from WSDL is the service element which is renamed to network in SIDL. The justification for this is that the word network is more suited in the context of system interoperability whereas service concerns a different domain.

3.5.1.1 Bindings and Protocols

A binding "describes a concrete message format and transmission protocol which may be used to define an endpoint" [28]. It defines the implementation details necessary to enable data exchanges on a bus. This realizes data exchanges because systems are connected on the buses through their ports.

A binding is declared using the **`binding`** element followed by its name. The bus detailed by a binding is specified using the **`of`** construct followed by the bus reference. The protocol to be used by the bus is specified using the **`as`** construct followed by the protocol reference. A binding declaration example follows in SIDL.

```
// An HLA 1516-2010 binding for RadarSystemBus
binding HlaBinding of RadarSystemBus as HLA.Protocol1516_2010
```

A binding allows the configuration of the protocol by specifying aspects such as encoding and QoS. Each protocol provides properties which can be configured instead of using the default values (refer to Chapter 4 for a more complete list). The following example demonstrates HLA and DDS bindings for the radar system (Figure 3-3).

```
// An HLA 1516-2010 binding for RadarSystemBus
binding HlaBinding of RadarSystemBus as HLA.Protocol1516_2010:
  // Configure aspects common to all channels
  channels:
    qos:
      Reliability = BestEffort

  // Configure a specific channel
  channel Detections:
    // Specify QoS for DetectionList.Items
    qos Items:
      Reliability = Reliable
      Order = Receive
      Sharing = PublishSubscribe

    // Specify type encodings
    encode RadarStateEnum as HLAoctet
    encode EntityIdentifyer as HLAfixedRecord

// A DDS 1.2 binding for RadarSystemBus
binding DdsBinding of RadarSystemBus as DDS.Protocol1_2:
  channels:
    qos:
      Reliability.Kind = BestEffort
      Durability.Kind = Volatile

  channel Detections:
    qos:
      Reliability.Kind = Reliable
      Durability.Kind = Transient
      History.Kind = KeepLast
```

Bindings can configure aspects common to all channels, through its **`channels`** property, or for specific ones, using the **`channel`** property as illustrated by the previous example. For instance, the `DdsBinding` uses a best-effort reliability kind which makes all channels use this default. The `Detections` channel changes this to require reliable communications instead. The **`qos`** `Items` of the `HlaBinding` configures the `Items` field of `DetectionList` to use reliable communications.

The author considers QoS as sensitive to interoperability and that is why it is modeled instead of being delegated to external configuration. Moreover, QoS is positioned in bindings because it is intrinsically protocol-related. The default encoding can be changed with **`encode`**. Thus, `RadarStateEnum` gets encoded as a single byte, in the `HlaBinding`, instead of four (Table 3-1).

Protocols are provided through model compiler extensions (Figure 1-4). This is, in essence, the same way WSDL [28] provides extensibility for service descriptions. Details regarding this support are provided in Chapter 4. Nonetheless, extensibility is an enabler to fully capturing and validating the details of system interface descriptions.

3.5.1.2 Networks, Endpoints, and Addresses

A network describes a set of endpoints at which a particular implementation of a bus is provided. In other words, a network provides protocol-bound data access to systems because it realizes the bus through a binding. The endpoint "associates a network address with a binding" [28]. From the endpoint's address, systems gain access to the bus. There, data exchanges become bound to the protocol and characteristics specified by the endpoint's binding. This results in systems being interconnected. A similar mechanism is used by WSDL [28] for web services. Service clients gain access to the service interface from a service endpoint that is bound to a protocol.

A network is declared using the **`network`** element followed by its name. The bus associated to the network is declared with the **`of`** construct followed by the bus reference. Endpoints are added to networks using the **`endpoint`** element followed by the endpoint's name. The endpoint's binding is specified using the **`of`** construct followed by the binding reference. The address of the endpoint can also be specified using the **`address`** element on the endpoint. It is optional as more often than not, the address is not known a priori. Additionally, as with protocol details on bindings, address details are protocol-specific and are discussed in Chapter 4.

The following example shows the radar system network which provides access to the `RadarSystemBus` from the bindings previously declared.

```
network RadarSystemNetwork of RadarSystemBus:
  endpoint Hla of HlaBinding:
    address:
      FederationName = 'Radar System'

  endpoint Dds of DdsBinding:
    address:
      DomainId = 0
      Partition = 'Radar System'
```

This completes the description of the radar system example (Figure 3-3). The resulting description captures all the relevant facets surrounding system interoperability. The next section presents how to use this information particularly for automation purposes.

3.6 Using SIDL Descriptions

The main purpose of SIDL descriptions, from the perspective of this thesis, is to automate the date-interchange software of systems. In order to accomplish this, a system implementation needs to refer to two SIDL elements in order to fully capture its interface and the details surrounding its data exchanges. The two SIDL elements are:

- System Reference
- Network Endpoint Reference

First, a system needs the definition of its interface as captured in SIDL which is achieved by referring to the corresponding `system` element. The system element links to the *Interface* facet as well as the *Data* facet since the system's ports refer to the data the system exchanges.

Second, a system needs the definition of *how* the data is exchanged, and from *where* it is accessible. This is achieved by referring to the `endpoint` of a `network` element. The endpoint specifies the access point, and its associated binding covers the transport details. Therefore, these definitions capture the required details concerned by the *Transport* facet. Moreover, because the binding associated on the endpoint captures the *Transport* details of a specific bus, *Connection* information becomes available.

Therefore, the data-interchange software of any system implementation can be completely derived from a SIDL system reference and a network endpoint reference. The following example demonstrates these two elements for the radar sensor using the HLA binding (Figure 3-3).

```
system RadarSensor:
  input Entities of Entity
  output RadarCrossSections of RcsList

network RadarSystemNetwork of RadarSystemBus:
  endpoint Hla of HlaBinding:
    address:
      FederationName = 'Radar System'
```

3.6.1 Data-Interchange Software Automation

The general methodology used to generate the data-interchange software of a system involves a two-stage workflow. The first stage, which is modeling, is used to model, in SIDL, the system interfaces capturing all the details related to the system interoperability facets. The second stage, which is code generation, involves specifying the system and endpoint in order to generate the corresponding software artifacts.

The author proposes a two-stage workflow in favor of a single-stage one. A single-stage workflow would see the compiler input SIDL descriptions and generate the corresponding software artifacts directly. In this context, SIDL descriptions, which are equivalent to source code, are reused directly instead of being shared through a library. The two-stage workflow ensures that SIDL descriptions, i.e., the source code, are only reusable through libraries

preventing the problems associated to reusing source code directly. The following sections describe these two stages in detail.

3.6.2 SIDL Modeling Stage

The modeling stage, illustrated in Figure 3-5, is used to create reusable SIDL libraries from SIDL descriptions. These libraries can be referenced by other SIDL descriptions to share common definitions, or used in the code generation stage to generate the software artifacts. A SIDL library is created by a model compiler which inputs SIDL descriptions and ensures their validity before generating the library. The model compiler is supported by a metadata library containing the SIDL element definitions. It is used by the compiler to encode the descriptions into a SIDL library providing common grounds for SIDL model compilers and the code generation stage.

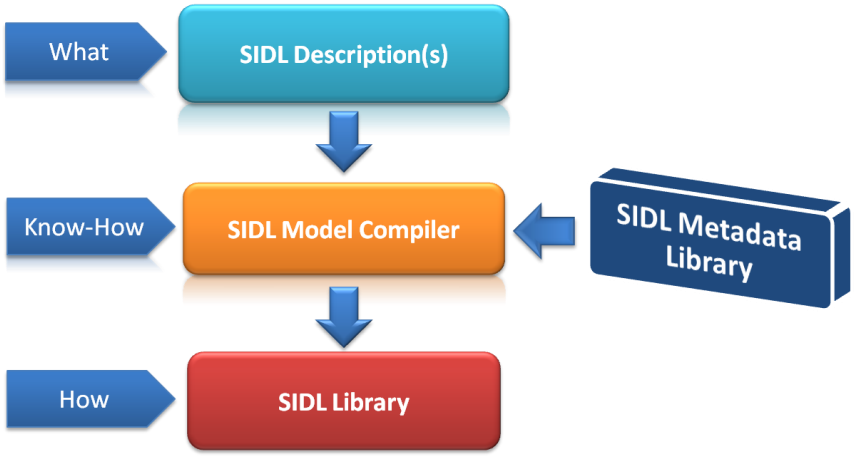

Figure 3-5: SIDL Modeling Stage

3.6.3 SIDL Code Generation Stage

The code generation stage, illustrated in Figure 3-6, is used to generate the software artifacts realizing the data exchanges of a specific system. The SIDL code generator inputs SIDL libraries and the reference to a system element as well as to the endpoint of a network element. Because code generation is largely contextual, the generator also inputs specific settings such as the output programming language of the software artifacts or preferences impacting the generated software interface exposed to the system using it. Additionally, the generator shares the same metadata library as used in the modeling stage simplifying the analysis of SIDL libraries. Frameworks and libraries can also support the code generation process. For instance, middleware dependencies

can be injected in the generated software artifacts. From there, the data-interchange software is generated and can be added to the system's implementation. Relying on external libraries enables easier software maintenance as the external library can be changed without requiring the system implementation to be rebuilt and deployed again. This step could also be done, in whole or in part, at runtime as is suggested by [60] in the context of dynamically generated gateways using runtime code generation.

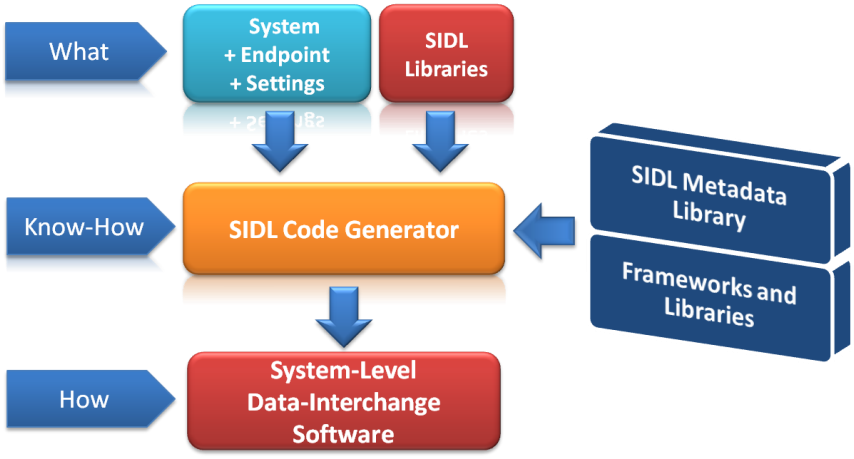

Figure 3-6: SIDL Code Generation Stage

A concrete implementation of the two-stage workflow is documented in Chapter 4.

3.7 SIDL Model Compiler Behavior

This section covers the rules which are not captured by the SIDL grammar (Appendix A) and which model compilers must enforce. These rules are required in order to make SIDL more portable and involves consistent compiler behavior.

3.7.1 Identifier Declaration Rules

- A new element's name must be unique in its namespace within its declared library.

3.7.2 Composition Rules

- Composition shall never resolve to circular definitions which are proscribed. This applies to the following elements: `entity`, composite `fact`, composite `measure`.

3.7.3 Fact Rules

- A field's name must be unique within a composite `fact`.
- A field's type must resolve to any other `fact` within a composite `fact`.

3.7.4 Measure Rules

- A simple `measure` cannot specify a `precision` when its representation is an `enum`.
- A simple `measure` can declare at most a single `units`, `frame`, and `precision`.
- A field's name must be unique within a composite `measure`.
- A field's type must resolve to any other `measure` within a composite `measure`.

3.7.5 Enumeration Rules

- An enumerator's name must be unique within an `enum`.

3.7.6 Array Rules

- An array's element type when specified as an identifier must reference a valid `entity`, `measure`, or `fact`.

3.7.7 Entity Rules

- A field's name must be unique within an `entity`.
- A field's type when specified as an identifier must reference a valid `entity`, `measure`, or `fact`.

3.7.8 Variant Rules

- A `case`'s value must be unique within a `variant`.
- An alternative's name must be unique within a `variant`.

3.7.9 View Rules

- A `select`'s member name, or alias when specified, must be unique within a `view`.

- A `select`'s type when specified as an identifier must reference a valid `entity`, `measure`, or `fact`.
- A `select` adapting a type with the `of` construct when both the referenced entity member's type and the specified type represent the same type reference shall result in a warning.

3.7.10 System Rules

- A port's name must be unique within a `system`.
- A port's type when specified as an identifier must reference a valid `entity`, `measure`, or `fact`.

3.7.11 Bus Rules

- A `channel`'s name must be unique within a `bus`.

3.7.12 Property Rules

- A `property` declaration must reference a declared property of the associated `protocol`.
- A `property` declaration referencing an identifier must reference a value declared by the associated `protocol`.

3.7.13 Binding Rules

- A `channel` must reference a declared channel on the associated `bus`.
- A `channel`'s name must be unique within a `binding`.
- A `encode` declaration must reference a valid `entity`, `measure`, `fact`, or `enum`.
- A `property` declaration affecting previously declared ones has precedence over them within the same scope (`channels` or `channel`) or for future references (`channels`).
- A `key` declaration must be unique within a `channel`.
- A `key` declaration must resolve to a valid field relative to the channel's associated message type.
- A `qos` declaration must be unique within a `channel`.

- A `qos` declaration must resolve to a valid field relative to the channel's associated message type.

3.7.14 Network Rules

- An `endpoint`'s name must be unique within a `network`.
- An `endpoint` declaration must reference a binding which is associated to the same `bus` as the `network`.

3.7.15 Unspecified Behavior

- A `view`'s `select` adapting a type with the `of` construct when both the referenced entity member's type and the specified type are different shall be implementation specific[3].

[3] See the Limitations in Section 6.3 for more details on this behavior.

Chapter 4 EXPERIMENTAL IMPLEMENTATION

This chapter focuses on the experimental implementation used to elaborate the SIDL language, the SIDL model compiler, and the SIDL code generator. Section 3.6 presented the general approach which involved a two-stage workflow. This chapter covers a concrete implementation of this workflow. Moreover, this chapter covers the implementation challenges faced while implementing both stages as well as the implementation choices which were made in the context of this experimentation.

This chapter is separated into two main sections each one covering a distinct perspective: workflow and system interoperability facets. The workflow perspective examines the various artifacts, their relationships to one another, and the technological choices supporting the two-stage workflow's implementation. The perspective of the system interoperability facets examines the implementation of each facet throughout the iterative methodology (refer to Section 2.5). Following these two sections is one describing the strategy used to validate the implementation.

4.1 Two-Stage Workflow

The workflow used to create the test applications involves two stages which are generalized in Section 3.6. The first one is to create SIDL descriptions. The second one is to generate the associated code. The reason for this is to allow the reuse of the elements declared in SIDL descriptions. For instance, a SIDL library contained only the data model elements. Other SIDL descriptions can reuse it to describe their specific system interfaces, data, connections, and transport.

4.1.1 Modeling Stage Implementation

Figure 4-1 presents the implementation of the SIDL modeling stage. SIDL descriptions are created using the SharpDevelop [69] text editor. The selection of this tool derives from the choice of the compiler baseline used to create the SIDL model compiler. The SIDL description compiler, i.e., model compiler, is embedded within SharpDevelop. It compiles SIDL descriptions into a SIDL library while validating them a priori.

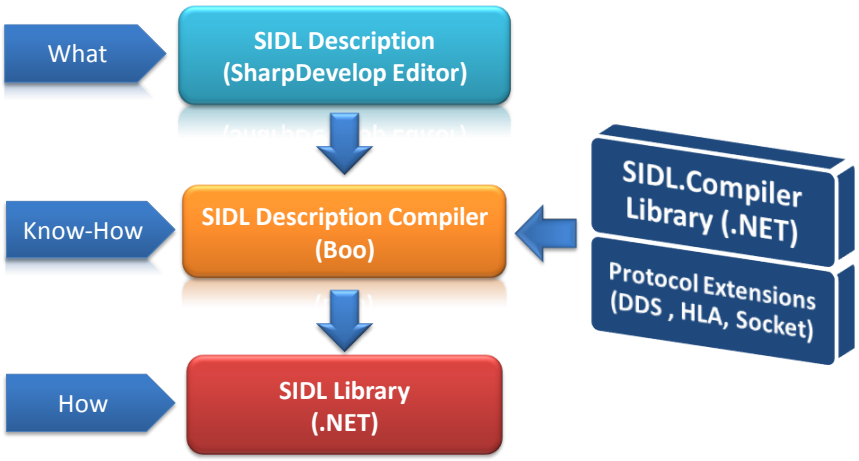

Figure 4-1: SIDL Modeling Stage Implementation

The SIDL model compiler itself is an extension to Boo [70]. Boo is an open source programming language for .NET platforms [71] featuring strong metaprogramming capabilities. It is particularly well suited to create DSLs [72]. It was selected over the Eclipse Modeling Framework [73] principally for its .NET support, simplicity, and integration with text editors, such as SharpDevelop, which can be embedded in other .NET application stacks. .NET support was mandatory in the industrial context where this research was held. Moreover, existing codebases were principally in .NET, and in C++ which .NET is particularly well suited to interoperate with. The goal of the compiler baseline selection was to select one which did not require writing a lexical analyser, a grammar parser, and a full-featured compiler stack from scratch including abstract-syntax tree analysis as well as error handling. Therefore, Boo was the most suited choice.

The SIDL compiler is defined within the *SIDL.Compiler* .NET library which is an extension to Boo. This library is also the implementation of the SIDL Metadata Library illustrated in Figure 3-5 when describing the general approach to the modeling stage. As a Boo extension, this library intervened within the Boo compilation stages in order to create the SIDL DSL. Therefore, this implementation makes SIDL an internal DSL [74] as the full Boo language features are still available to SIDL modelers. This limitation can be addressed by changing Boo's parser stage to make SIDL an external DSL [74] therefore only presenting SIDL constructs to modelers. This work was left outside the scope of this research as it does not impact the demonstration of the research objectives, moreover the elaborated SIDL models only use the SIDL grammar. Nonetheless, future work can make an external DSL out of SIDL with Boo.

Boo outputs .NET libraries which, therefore, are used by the SIDL compiler to represent SIDL libraries. The SIDL compiler transforms SIDL descriptions and maps them to a common metadata interface with the help of *SIDL.Compiler*[4]. For instance, this enables a `network` in a SIDL description to reference a `binding` in another SIDL description as well as a `bus` in a referenced SIDL library. This methodology is the same one used for typical software development with the exception that the source code is in SIDL. This enables configuration management with strong versioning as .NET libraries exhibit strong-names [75].

Furthermore, one could use the *SIDL.Compiler* library directly to create SIDL descriptions from C#, or any .NET language, since the library represents all SIDL elements which are used by the SIDL model compiler. This mechanism is used to expose a `protocol` to a `binding` through SIDL compiler extensions. This facilitates the integration of new protocols as they are left outside of the SIDL compiler, and can be written in any .NET language. Moreover, protocols implement an interface defined by *SIDL.Compiler* as any other SIDL element. In the context of the research, the following protocols were implemented, using Boo, as SIDL compiler extensions: DDS [34], HLA [10], and a generic Socket protocol for UDP communication. The Socket protocol is used to support the implementation of DIS [6].

Additionally, a protocol exposes a list of properties which can be modified by bindings. For instance, the DDS protocol exposes a `Reliability` property for the `qos` of a `channel`. This value is defined as an enumeration, containing `BestEffort` and `Reliable` choices, and provides the valid range of values the SIDL compiler can accept. The same pattern applies to an endpoint `address` and `encode` which are protocol-specific and extended the same way as `qos`.

4.1.2 Code Generation Stage Implementation

With SIDL libraries created from SIDL descriptions, a `system` and an `endpoint` can be specified to generate code from them as illustrated in Figure 4-2. The code generation consists of transforming the specified inputs into C++ or C# code based on the settings passed to the generator.

---

[4] For brevity, the *SIDL.Compiler* API is not described in this thesis, but can be provided on demand by the author.

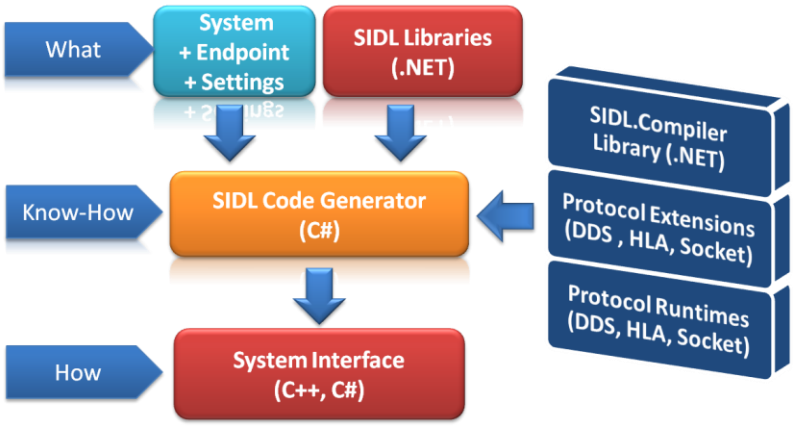

Figure 4-2: SIDL Code Generation Stage Implementation

The code generator reuses the *SIDL.Compiler* library to share the core definitions of all the SIDL elements used by SIDL libraries. This greatly reduces code generation complexity because the metadata is accessed as strongly-typed SIDL elements rather than as generic metadata or text. For instance, the `RadarSystemNetwork` of Section 3.5.1.2 can be accessed as an `INetwork` object as declared by *SIDL.Compiler*. Additionally, one important aspect of the *SIDL.Compiler* library is that it allows for a SIDL library to be re-created from a SIDL description. That is because a SIDL library presents the same metadata as a SIDL description. Therefore, it is possible to re-create a complete SIDL description from a SIDL library.

The strategy of sharing the *SIDL.Compiler* library between both stages supports well the iterative nature of the methodology used (refer to Section 2.5). Extending the language with new concepts only impacts the compiler until the code generator is ready to use them. This also makes updating the code generator trivial when the compiler changes since breaking changes are easily pinpointed in either stage because of the strongly typed information used.

On the code generation side, T4 text templates [76] is used to etch-out the target code skeleton and fill it based on embedded C# code constructs. Code generation patterns can be used with T4 which simplifies even more this task. As an example, HLA requires the generation of code which handles code serialization and de-serialization. Because both exhibit the same code generation structure, a pattern is used handle their common structure while specific implementations fill it with either serialization or de-serialization code snippets.

The generation of the software artifacts is supported by runtime libraries notably ones simplifying protocol handling and encoding. This simplifies code generation because the libraries

encapsulate complexity that is not required to be generated. For instance, generating the DDS QoS attributes requires the interaction of many objects which the DDS runtime library simplifies. The following generated C# example demonstrates this.

```
// With library support
var qos = participant.TopicQosBuilder()
  .Reliable()
  .Transient()
  .KeepLast()
  .Shared()
  .Build();

// Without library support
DDS.TopicQos qos;
participant.GetDefaultTopicQos(ref qos);
qos.Reliability.Kind = DDS.ReliabilityQosPolicyKind.ReliableReliabilityQos;
qos.Durability.Kind = DDS.DurabilityQosPolicyKind.TransientDurabilityQos;
qos.History.Kind = DDS.HistoryQosPolicyKind.KeepLastHistoryQos;
qos.Ownership.Kind = DDS.OwnershipQosPolicyKind.SharedOwnershipQos;
```

Generating the software artifacts starts by finding the system and endpoint within the SIDL libraries provided as input. Then, based on the type of source code to generate, a code-specific factory is selected. Additionally, based on the type of protocol required by the endpoint, as defined by its associated binding, a protocol-specific factory is selected. Then, both factories work hand-in-hand to generate the software artifacts. For instance, the HLA factory generates the HLA FOM module required by HLA runtimes. Another example is with DDS which requires an IDL file to be generated. The IDL file is then used by DDS implementations to handle data serialization. The following example demonstrates a SIDL description with its corresponding IDL representation.

```
//
// SIDL
//
enum State:
  StandBy
  Start
  Stop

entity Controller:
  Id as long
  State as State

system ControllerSystem:
  output Controllers of Controller

bus ControlBus:
  channel Controllers of Controller

binding ControlBusDdsBinding of ControlBus as DDS.Protocol1_2:
  channel Controllers:
    key Id

network ControlNetwork of ControlBus:
  endpoint Dds of ControlBusDdsBinding
```

```
//
// IDL: Generated using above SIDL, System: ControllerSystem, Endpoint: ControlNetwork.Dds
//
enum State
{
  StandBy,
  Start,
  Stop
};

struct Controller
{
  unsigned long long Id;
  State State;
};
#pragma keylist Controller Id
```

4.2 System Interoperability Facets Implementation

Throughout the implementation of the system interoperability facets, many challenges emerged. For instance, while elaborating the SIDL language, syntax changes would trigger the most source code refactorings. Introducing new concepts were less impacting, except changing the language would have ripple effects down to the systems using the generated software interface. Therefore, the following sections present how SIDL was implemented covering the language, the compiler, and the code generation with reference to the system interoperability facets.

4.2.1 Data Facet Implementation

The *Data* facet was the first one to be implemented as data definitions represent the foundation of any data exchange. The language side was implemented based on the FACE data model [26] with the exceptions detailed in Chapter 3. This part of the language was the simplest primarily because of the abundance of data exchange models. On the code generation side, this would consist of elaborating the target form of the code required by the test applications, and then automating it. The data model definitions would cover all of the data model elements, e.g., observables, measures, informations, facts, entities, variants, views... They would all be transformed into .NET classes which inherit from a base interface corresponding to their SIDL element counterpart in *SIDL.Compiler*. An entity inherits from `IEntity`, simple measure from `ISimpleMeasure`, and so on.

Each iteration of the language would introduce new data elements, revised by SMEs, which the code generator would transform. For instance, the introduction of measures broke the code generation because entity fields were now missing from the generated code being modeled as

entities prior to the language update. Moreover, some iterations were used only to refine the generated code because of impacts it would have on systems using it. For instance, simple measures were transformed into C++ structures at first with a single field representing its value. Except this would impact the performance of the DDS middleware which was optimal when replacing structures with their corresponding value type. In other words, the generated software interface used a `double` for a speed measure instead of a speed structure containing a `double`. Preserving type-safety in the generated code would have been ideal, except it was left outside of this research not impeding the demonstration of the research objectives. Moreover, these changes originated from recommendations of the middleware vendor.

The greatest challenge encountered while implementing the *Data* facet was with `view`s. This required special handling on the Boo pipeline because a `select` statement might reference metadata that is currently being compiled in another SIDL description. This metadata was not yet available while a `view` was being processed by Boo. Therefore, a handler was attached to the stage of the Boo compilation pipeline which made this metadata available[5]. This allowed views to be properly generated in the output SIDL library, and validated before doing so. Unfortunately, view support was limited to representing entities due to the lack of conversion support. This limitation is further discussed in Chapter 6.

4.2.2 Transport Facet Implementation

Implemented next was the *Transport* facet in order to link the data model to protocols, and enable preliminary data exchanges. At this point, bindings were simplified, even from their WSDL counterpart [28], in order to consider the data model directly instead of buses. This was revisited when implementing the *Connection* facet with the help of SMEs on the language side. Protocol implementation was facilitated by the use of extensions on the SIDL description side to seamlessly introduce new protocols, and by protocol runtimes on the code generation side.

For DDS support, the data model was transformed into IDL which DDS implementations use for data serialization through their own code generation (IDL to C++ and C#). To simplify code

[5] The AfterStep event of the Boo compilation pipeline is used waiting for the ResolveTypeReferences to complete.

generation, DDS helper classes were created in a helper library. This would also reduce the size of the generated code as common source code was now shared in the helper library.

A socket was also implemented to support a reduced DIS data model. Its elaboration was more challenging because of the lack of higher-level functionality, as offered by DDS or HLA, and it required the generation of the code to encode data which was automated by DDS. In order to simplify code generation, a dedicated serialization library was created incorporating stream readers and writers. The streams also simplified the handling of endianness through swapping and non-swapping classes, i.e., `StreamWriter` and `StreamWriterSwapped`. DIS expects data to be serialized in big-endian ordering therefore the target application decided to either use the swapped or non-swapped version at runtime based on the execution platform's network ordering.

HLA support was the most challenging because of the additional code generation required for generating the HLA FOM module. HLA applications use FOM modules for data exchanges at runtime whereas DDS generates the required metadata at compile-time. Additionally, HLA required the generation of data encoding similar to the socket implementation with the main difference that data alignment rules exist for predefined encodings. HLA encoding helpers were not used for FOM encoding because of their reliance on dynamically allocated memory [77]. Nevertheless, they would only have simplified the data alignment rules of the predefined encodings [17], e.g., `HLAfixedRecord`, `HLAvariantRecord`. As with the DDS and socket implementations, helper classes were used to simplify code generation.

As with views in the *Data* facet, bindings were the greatest challenge encountered while implementing the *Transport* facet because they require metadata that is not yet available when being processed. The same trick was used to attach a handler to the Boo compilation pipeline and wait for the right stage to provide the required metadata.

4.2.3 Interface Facet Implementation

Next to follow was the *Interface* facet's implementation. This required representing systems and their ports in SIDL libraries which were trivial as it only required the addition of new SIDL element metadata. Unfortunately, in order to implement the code generation side, the *Connection* facet had to be implemented first because no link existed at this point between data, transport, and interfaces.

4.2.4 Connection Facet Implementation

The implementation of the *Connection* facet was the most challenging and required a major refactoring of the compiler, the code generator, and the test applications. That is because it is the glue between all the other facets dealing with the full stack from the system interface down to the transported data. The compiler side was the most challenging because of the introduction of the bus element and the connections of system ports to bus channels. Because the language elements were based on AADL [40], this provided guidance for their introduction.

In the early stage of implementation, the code generator referenced bindings directly to trigger code generation. Bindings provided the link to data directly since buses were not yet introduced. The introduction of the bus element broke how data was provided to the code generator therefore triggering a major refactoring. This resulted in the analysis of the bus channels based on the selected system to determine which data to use by correlating the connected ports. Moreover, representing the system's ports in the generated code consisted of generating readers and writers in the software interface. At the same time, this completed both the *Interface* and the *Connection* facet implementations.

4.2.5 SIDL to DDS Mapping

The SIDL to DDS mapping consists in generating a DDS participant in the software interface. A DDS topic is generated for each `bus channel` using the same `channel` name for the topic. Each topic's QoS attributes are mapped from the corresponding `qos` declarations in the corresponding `channel` declaration, or `channels`, in the `binding`. The `endpoint`'s `address`, when specified, sets the DDS domain identifier and partition name of the DDS participant. The generated IDL file includes all the types referenced by connected channels recursively including their inner declarations therefore capturing the complete data exchange model. For instance, fields of entities are recursively analysed to include their declarations. The mapping of SIDL types to IDL types is presented in Table 4-1. The mapping of value types is based on the size of the SIDL's representation size (Table 3-1) when no direct correspondence exists in IDL.

Table 4-1: SIDL to IDL Type Mapping

SIDL Type	IDL Type
`Entity`	`struct`
`View`	`struct`
`Variant`	`union`
`Enum`	`enum`
`Composite Fact`	`struct`
`Simple Fact`	`IDL value type based on representation`
`Composite Measure`	`struct`
`Simple Measure`	`IDL value type based on representation`
`Bounded Array`	`array`
`Unbounded Array`	`sequence`
`Value Type`	`Corresponding IDL value type, otherwise based on size`

4.2.6 SIDL to HLA Mapping

The SIDL to HLA mapping consists in generating a HLA RTI federate ambassador in the software interface. For each **`bus channel`** that is connected, when an encoding is specified as `HLA.interactionClass`, then an HLA interaction class with the channel's type name is generated, otherwise it is an object class. Each field of the channel's type is considered a parameter in the case of an interaction, or a class attribute otherwise. Each topic's QoS attributes are mapped from the corresponding **`qos`** declarations in the corresponding **`channel`** declaration, or **`channels`**, in the **`binding`** to the parameters/attributes. The mapping of the type of a parameter/attribute is based on Table 4-2. The mapping of value types is based on the size of the SIDL's representation size (Table 3-1) when no direct correspondence exists in HLA. Unsigned integer types have been added to the set of basic data types as HLA only provides signed versions. This lack should be addressed within the next revision of the standard.

Table 4-2: SIDL to HLA Type Mapping

SIDL Type	HLA Type	Default HLA Encoding
Entity	fixedRecord	HLAfixedRecord
View	fixedRecord	HLAfixedRecord
Variant	variantRecord	HLAvariantRecord
Enum	enumeratedData	HLAinteger32BE
Composite Fact	fixedRecord	HLAfixedRecord
Simple Fact	simpleData	Fact's representation
Composite Measure	fixedRecord	HLAfixedRecord
Simple Measure	simpleData	Measure's representation
Bounded Array	fixedArray	HLAfixedArray
Unbounded Array	variableArray	HLAvariableArray
Value Type	Corresponding HLA basic type, otherwise based on size	Determined by selected HLA basic type

The `endpoint`'s `address`, when specified, sets the HLA federation name. The generated HLA FOM module is constructed in the same fashion as the DDS IDL file with the exception that it is an XML file and its content is based on Table 4-2. Moreover, when an encoding is specified, it is used instead of the default encoding as described in Table 4-2.

4.2.7 SIDL to DIS Mapping

The SIDL to DIS mapping consists in generating a socket participant, as defined in the socket helper library, in the software interface. A socket topic is generated for each `bus channel` using the same `channel` name for the topic. The generated data exchange model is constructed in the same fashion as the DDS IDL file, as described in Section 4.2.5, with the exception that it is in C++ or C#.

4.3 Implementation Validation

As presented in the iterative methodology (Section 2.5), the success criteria of each iteration involved the demonstration of functional test systems exhibiting the same behavior as the one defined in the previous iteration.

Testing the compiler involved the creation of test cases written in SIDL using all of the language elements. This would ensure that regressions on the language were captured by these tests. Testing the compiled SIDL libraries against compiler regressions was found to be the most complex task as the only decent methodology found was to use the code generator. This involved ensuring the code generator would cover all of the metadata exposed by the SIDL library. This proved to be challenging as language elements were introduced faster than they were used by the code generator. Nonetheless, this strategy caught many breaking changes in the SIDL library compilation and typically involved major metadata changes.

Because the focus of this research is on data exchanges, regression tests were used to validate their behavior. The performance aspects of the generated code were not examined as it replicated what was considered to be the optimal code that a senior software developer would have manually written. This was also inherent to the iterative methodology used as the code generator was built against code templates developed by such software experts.

The first kind of regression tests involved the comparison of the generated software artifacts with the manually created ones. This would ensure that the code generation preserved the same software elements. It was often required to reformat the manually created source code to enable better comparison. These tests were applied to both C++ and C# code bases independently primarily because of the inherent differences in the middleware and helper library interfaces. This methodology proved to be very effective as many errors were found this way before going any further.

The second kind of regression tests consisted in interoperating a test system using the updated software interface with the same system using the software interface of the previous iteration. Moreover, these tests were done manually using data injection from user interfaces. Because the comparison tests validated that the same software characteristics were preserved, the interoperability tests only consisted in visually determining if data was exchanged or not. This proved to always be sufficient as the identified regressions only came from errors introduced in the supporting libraries and the protocol runtimes. Once deemed valid, the entire code base was updated to only preserve the newly created software artifacts.

All of these regression tests have therefore demonstrated the experimental implementation. The following chapter focuses on the results obtained using this experimental implementation.

Chapter 5 EXPERIMENTAL RESULTS

This chapter presents the experimental results of using SIDL to address specific test cases. Each test case is a distributed software application involving the interoperability of test systems. They are detailed with a particular regard over their data-interchange software which is generated from SIDL descriptions. All the test cases and the test applications are detailed in this chapter.

The primary goal of all the test cases is a contract-first approach to the data exchanges. That is, any data exchanged by the test systems needs to be specified in SIDL. This ensures that special cases are always dealt with in SIDL instead of being compensated for in the application's codebase. Moreover, this chapter presents the experience of SMEs modeling system interfaces and the details surrounding their data exchanges in SIDL, along with using the generated software artifacts emanating from these models into their code base.

5.1 Test cases

From the start of the experimentation phase of this research, test cases were elaborated in order to capture the essence of the targeted system integration and interoperability issues emanating from multiple discussions and refinements with SMEs along with literature data. These culminated into the research's objectives and resulted in the following test cases. The full SIDL descriptions used to capture these test cases is provided in Appendix B.

5.1.1 Test Case 1 - Colliding Balls

This test case involved reusing a simple distributed simulation application which basically involves balls colliding together. The goal of this test case is to have a representative system that SMEs would develop dealing with the system's inputs and outputs, i.e., its data exchanges. It is meant to demonstrate the software interface exposed to SMEs as well as their automation from SIDL descriptions.

In this test case, the balls, represented as rigid spheres of various sizes, bounce in a cubical room using a frictionless gravity-based model. Each instance of the application, denoted as a collision system, shares its data with the existing ones by distributing it over a communication middleware, DDS or HLA. This results in a distributed simulation of colliding balls where each collision system inputs and outputs balls. Moreover, a collision system extrapolates the positions

of balls and only computes the collisions of the balls it produced. In other words, local balls collide with local or remote balls from the perspective of a collision system. Because it is a distributed application, the extrapolation model used is the DIS_DRALG_DRM_RVW(4) of the dead reckoning algorithm [6] which at the same time determines the criteria for outputting data updates. The resulting test application is illustrated in Figure 5-1.

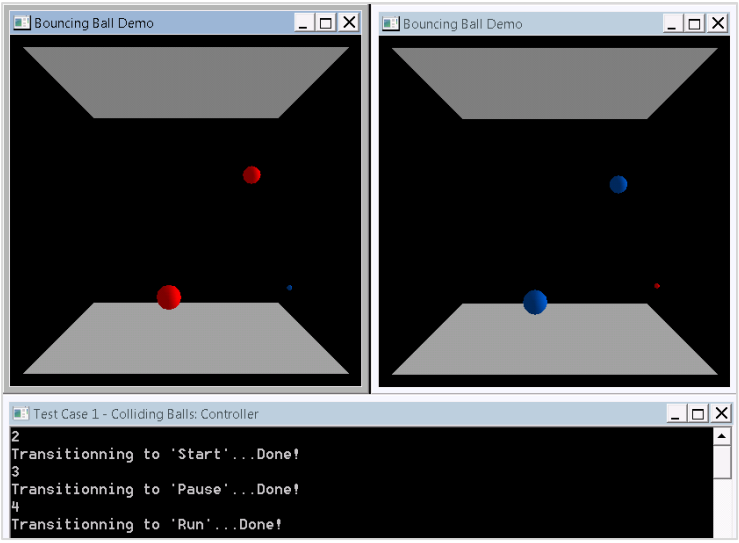

Figure 5-1: Colliding Balls Test Case

Figure 5-1 shows two test systems, the left one producing a single small ball while the right one produces two big ones, and a controller system. Each test system contains a 3D display where balls colored in blue are outputted by a system while the red ones represent inputted ones. The controller system is used to control the execution of the whole distributed application by outputting common tick and simulation times as well as a shared simulation state covering the start-up, pause, run, and stop states. Once the controller is in the run state, the collision systems execute their model. The collision system's user interface supports many options including the injection of new balls and the ball count at each injection.

On the architecture side, an *Entities* data model describes in SIDL the data that is exchanged between systems. A *Control* data model does the same for sharing state and time data. A collision system is loaded by a container which handles its execution. This way, collision systems do not need to be aware of the control's data exchanges and the containers can load any type of model, through a common interface, which has been used for all test case systems. All the code bases are in C++ for this test case.

5.1.2 Test Case 2 - Ownership Transfer

This test case is meant to demonstrate the concept of ownership transfer which implies having a system drive the value of another one such that the other systems see the new values instead of the original ones. Ownership transfer is a typical function used in distributed simulations. The test case reuses the collision system previously defined. It consists in having a ball follow another. To this end, a web front-end allows the selection of master/slave pairs. Once the ownership is assigned, the ownership system drives the position of the slave with the master's position offset by a constant. This is illustrated in Figure 5-2.

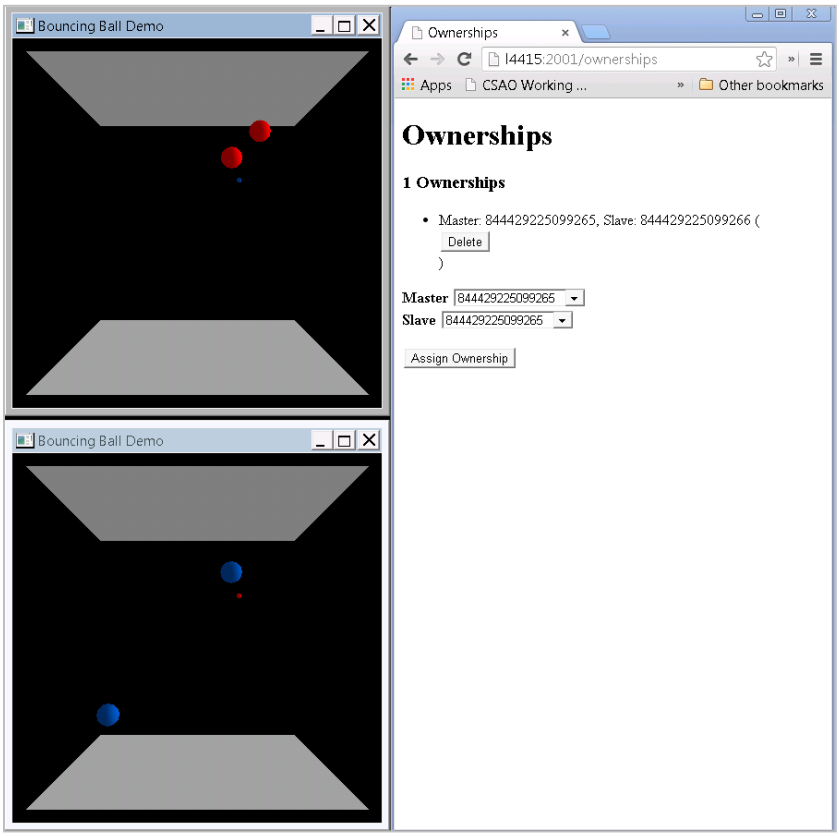

Figure 5-2: Ownership Transfer Test Case

One can notice that the top-most system sees the effect of the ownership while the bottom one does not. That is expected as this type of ownership is only visible at the middleware-level which is DDS here. The ownership system writes the position of the slave using a higher strength QoS therefore achieving the desired effect.

On the architecture side, the web front-end communicates with a C# REST service [78]. A SIDL ownership service is defined to input the ball descriptions and output the ownership data to the ownership system. Then, the ownership system, also defined in SIDL, outputs the new position data thereby demonstrating the ownership transfer. Moreover, an *Ownership* data model describes in SIDL the data exchanged between the ownership service and system.

5.1.3 Test Case 3 - DDS-DIS Gateway

This test case is meant to be representative of multi-architecture environment considerations by demonstrating a gateway application bridging two different architectures, DDS and DIS. The gateway bridges the *Entities* data model, used by the Colliding Balls test case, and a DIS subset modeled in SIDL. It is illustrated in Figure 5-3.

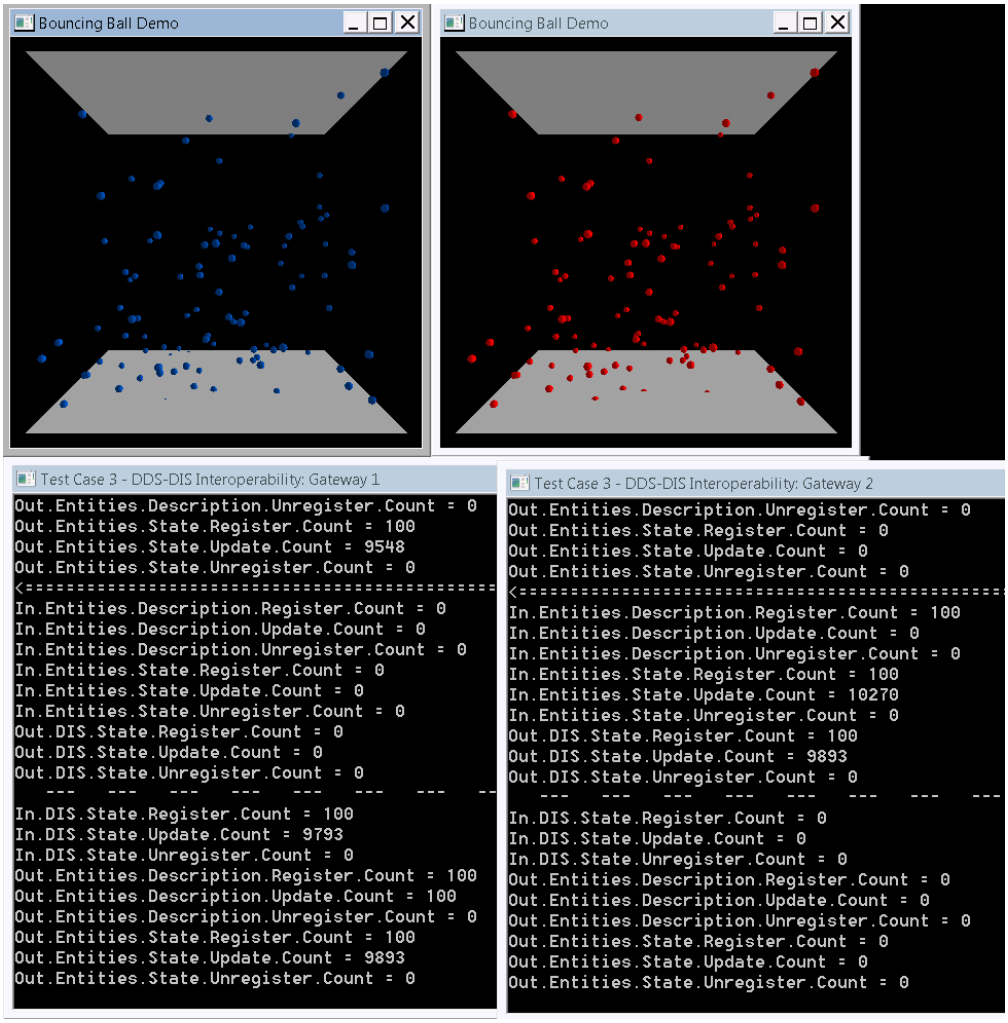

Figure 5-3: DDS-DIS Gateway Test Case

In order to reuse the collision systems, two gateway systems are used which share the same SIDL definition. Basically, each collision system resides in its own SIDL network. This is achieved by using different DDS partition names. In Figure 5-3, the left collision system resides in the *Entities* partition while the right one in the *Entities2*. In order to bridge both partitions with the two gateway systems, each gateway connects to either the *Entities* or *Entities2* partition. Then, each gateway system outputs the resulting DIS data on the same DIS network. Because each gateway system inputs DIS data from the same DIS network, this triggers the reverse transforms resulting in the collision system of the opposing network to receive the transformed balls. Therefore, the full data flow as seen in Figure 5-3 is as follows:

- Left collision system outputs to *Entities* network
- Left gateway system inputs from *Entities* network, outputs to *DIS* network
- Right gateway system inputs from *DIS* network, outputs to *Entities2* network
- Right collision system inputs from Entities2 network.

In addition, the gateway's code base is in C#.

5.2 Modeling System Interface Descriptions

The SME feedback from modeling system interfaces in SIDL is very positive. Its strongest highlighted force is its expressiveness. SMEs find that SIDL only shows them what they need to. Its separation of concerns also contributes to their understanding of the data models particularly as demonstrated by the SIDL descriptions developed for the test cases (Appendix B) which organizes SIDL descriptions by system interoperability facet (Figure 2). Moreover, most SMEs do not want to be aware of the transport technicalities which can be encapsulated in isolated SIDL descriptions and be taken care of by system integrators.

Another noted plus by SMEs is in regards to treating SIDL descriptions as source code. The SMEs used the same software tools to manage the SIDL files, notably the same revision control system and comparison tool, which made them "feel at home". They noted that this particularly contributed to their understanding of the evolution of the data models. The SIDL models are also stored within the same code base as the test systems.

The principal difference on the tooling side SMEs faced is the additional editor introduced to edit SIDL files. SMEs would have preferred to use a single tool for handling all of their source code. This tool integration concern was expected, but was left out of the scope of this research not hindering the demonstrability of the research objectives, and can be addressed by extending the existing source editor. Additionally, in order to link the SIDL code generation with their code base, their source code project had to be modified to include pre-build events. This consisted in providing to the code generator the SIDL system and endpoint references, the SIDL library references, and the target language. Again, SMEs would have preferred a better integration, but was required only once.

On the language side, the only criticized aspect of SIDL by SMEs was the array syntax which uses parentheses instead of the common square brackets. Moreover, SMEs asked for the inclusion of lower bounds such that they are able to capture lists with a minimum item count. This is also in line with UML modeling which presents cardinality has having both upper and lower bounds.

SMEs expressed that the validation messages of the SIDL compiler really helped them understand and correct their modeling errors. An example of this was the error messages which exactly pinpointed to the model elements impacted by the introduction of breaking changes, for instance, such as when renaming an `entity` as seen in Figure 5-4.

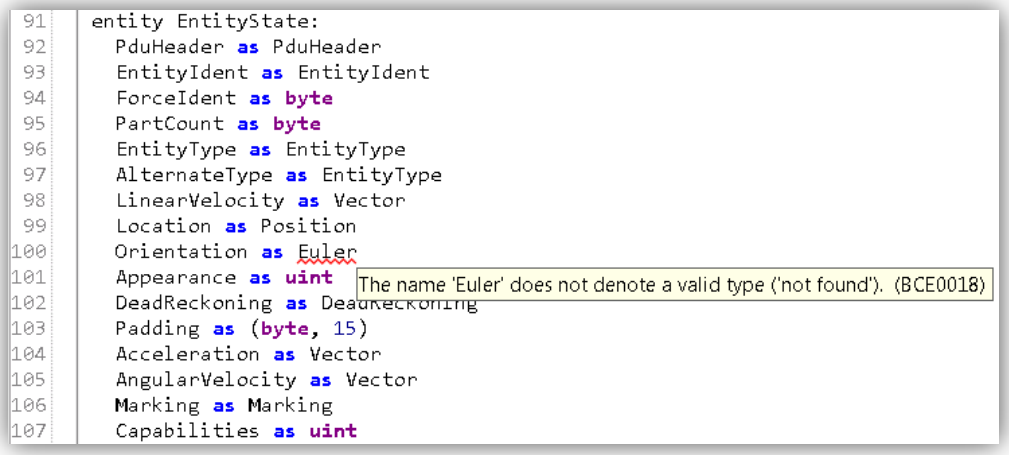

Figure 5-4: Pinpointing a Breaking Change in SIDL

Another example was the notification of the duplication of named elements particularly across multiple SIDL descriptions. Users even requested additional validation rules to further prevent

their typical modeling errors and increase their efficiency. An example of this was the validation omission of paths specified in the `select` clauses of a `view`. Errors were always captured, except it was by the compiler of the generated code instead of the SIDL model compiler. This complicated the user's understanding of the error and was hard to relate to the invalid `select`. The SIDL model compiler contextualized such errors.

The code generation capabilities have also been noted as enabling efficiency as it prevented typical encoding inconsistencies with the data model. Another noted plus was the reuse it enabled. Many data model refactorings occurred simply to remove duplications, particularly expressing the same concept except with different names, and to standardize specific data types as well as units.

One side to improve upon, as noted by users, is code completion, or autocomplete, which has not been implemented as of this writing. The tooling support is present, but the lack of time prevented its integration. The same applies to syntax highlighting which was partially implemented. SMEs considered it less important than code completion as they consider the language readable as it stands. The following example illustrates the targeted syntax highlighting versus the actual one.

```
// Expected syntax highlighting
measure Position_Meter_Double of Position as double:
  units Meter
  precision 0.00001

// Actual syntax highlighting
measure Position_Meter_Double of Position as double:
  units Meter
  precision 0.00001
```

One major benefit of SIDL, as highlighted by SMEs, was the ability to capture multi-architecture peculiarities by modeling them in SIDL in a uniform way. They noted that it was particularly useful while developing the DDS-DIS gateway as every aspect could be captured from a single viewpoint providing an architectural oversight enabling efficient access to the whole application. Moreover, it was noted that the configurability of SIDL networks easily helped reuse the same applications throughout the test cases without modifying them. For instance, the colliding balls test case could have its endpoint address configured externally allowing it to be reused easily for the gateway test case.

All in all, the experience of modeling system interfaces in SIDL was very effective and allowed SMEs to focus on their expertise by hiding software complexity from them. The following

chapter elaborates on this as it presents a general discussion over the research questions and the demonstration of the research objectives.

Part III

CONCLUSIONS

Chapter 6 GENERAL DISCUSSION

The preceding chapters aimed at contributing to the answer of the research questions, notably: a) how to capture system interfaces and which elements should be captured, b) how to capture multi-architecture considerations, and c) how to use system interface descriptions to automate system interoperability tasks. In this regards, the general methodology proposed to address specific research objectives. This led to advent of: 1) a new system interface description language used to capture system interfaces and the various aspects surrounding their data exchanges, and 2) a new method for automating the system-level data-interchange software from system interface descriptions. These new tools contribute to the simplification of system integration and interoperability. The following sections present a general discussion regarding these advances, particularly with reference to the current state of the art, focusing on their implications and limitations.

6.1 System Interface Description Language

This thesis studied three key aspects relating to system interfaces: the relevant language elements, modeling system interfaces with the language, and capturing multi-architecture considerations.

6.1.1 Relevant Language Elements

The first aspect studied involved the creation of a new perspective on system interoperability by introducing the system interoperability facets notably the system *Interface*s, the *Connection* of these interfaces to data, the *Data* exchanged between systems, and the data's *Transport* from system to system. Prior to finding the facets, the Levels of Conceptual Interoperability Model (LCIM) defined by [79] was the only architecture viewpoint overlooking system interoperability, and was only used to characterize the attainable levels of interoperability between systems.

On the other hand, the system interoperability facets structure the problem domain enabling a clear separation of concerns, and provide the architectural foundation to a system interoperability taxonomy. To this end, existing taxonomies were studied principally originating from standards. It was discovered that no single solution covers the full scope of the interoperability facets. Therefore, the language elements relevant to describing system interfaces were forged from existing ones, along with novel additions, into this new taxonomy in order to cover the full

spectrum of the system interoperability facets. In turn, this new taxonomy enables a better understanding of the elements impacting system integration and system interoperability, and the common language which can be shared amongst stakeholders, such as integrators, suppliers, and system experts.

6.1.2 Modeling System Interfaces

The second aspect studied involved modeling system interfaces using this new taxonomy. SIDL, the System Interface Description Language, would become this incarnation as described throughout Chapter 3. As demonstrated by the study of existing formats, i.e., meta-models, they primarily focus on *Data* with most ones only capturing this facet, exhibit limited validation semantics, or are not easily understandable by the stakeholders targeted by this research even if some languages are expressed in human-readable formats. The study identified DSLs as potential candidates for capturing system interfaces and addressing these issues.

Consequently, SIDL was materialized as a textual DSL covering all the system interoperability facets. Moreover, SIDL simplifies change identification as well as the understanding of interface evolution by being in the language of its stakeholders as demonstrated in Chapter 5. This is critical in finding issues early on in the development and integration processes. At the same time, SIDL provides a common interchange format which facilitates communication between stakeholders, simplifies interface governance, enables reuse of system interface descriptions, and provides common grounds for engineering tools. The impacts of the introduction of a new language to SMEs are counter-balanced by the fact that the language is designed to reflect their domain as such decreases its learning curve. Furthermore, managing SIDL descriptions as source code has the added benefit of leveraging the same software engineering practices including reusing the same revision control system and configuration management. For SMEs, this also hides complexity from them, increase their productivity, and better leverage their expertise.

6.1.3 Capturing Multi-Architecture Considerations

The third aspect studied focused on capturing multi-architecture considerations. Identifying a way to capture such detail with the right level of abstraction is a key problematic area of system integration and system interoperability as no solution currently exists. The study revealed that there is a general consensus towards an architecture-agnostic format to address these

considerations. Unfortunately, only preliminary work has been done, until now, in this area. Moreover, these research initiatives focus on simplifying gateway solutions which bridge multi-architecture environments therefore do not tackle the problem at its root. Nevertheless, they do recognize the need to have at least architecture-neutral representations of the data exchange models.

Consequently, SIDL was designed as an architecture-agnostic format used to capture, not only *Data*, but the full span of the system interoperability facets. Architecture-specific details are modeled in SIDL through its *Transport* elements as detailed in Chapter 3. Moreover, architecture-specific representations can be derived from SIDL descriptions as demonstrated in the experimental results presented in Chapter 5 particularly when tackling the DDS-DIS gateway. This implies that SIDL captures the details relevant to system data exchanges down to specific architectures. At the same this, this simplifies the introduction of changes to system interfaces as changes are automatically propagated down each architecture's own representation being introduced from a single architectural viewpoint defined in SIDL. As such, SIDL is an architecture description language. Additionally, capturing multi-architecture considerations in SIDL is an enabler to the further automation of system integration and interoperability activities.

6.2 Automation of the System-Level Data-Interchange Software

This thesis studied the automation of the software responsible for system data exchanges as a way of simplifying the tasks involved in system integration and interoperability. To this end, code generation techniques were studied and it was found that they are used extensively in the industry principally for automating software artifacts, such as source code and configuration data. Moreover, some solutions use these techniques in order to automate data serialization. Additionally, the study of DSLs revealed that their usage could allow SMEs to limit their work on the interconnection of system interfaces as well as the data they exchange instead of the intricacies of data serialization and communication protocols.

This led to the introduction of the SIDL code generator described in Chapter 4. The SIDL code generator has the novelty of generating the data-interchange software from system interface descriptions covering all the system interoperability facets. Therefore, this enables the code generator to take better decisions because it has access to a richer pool of information, notably from the high-level system relationships down to the low-level protocol and encoding details.

Because multi-architecture considerations are captured natively in SIDL, this enables the code generator to be architecture-agnostic making it reusable in other contexts.

The study of code generation techniques also found that they can be optimal in order to adapt to the environment in which the generated code is executed. This capability is mandatory in order to provide the flexibility required by systems to enable their reuse across multiple platforms in support of product lines. Consequently, the SIDL language was augmented based on these findings by introducing the concept of views in order to capture system interface variability, as described in Chapter 3, and enable their automation as described in Chapter 4.

The general methodology of transforming system interface descriptions into data-interchange software therefore culminates in the combination of the modeling workflow and the code generation workflow, i.e., the two-stage workflow presented in Chapter 3. This enabled the automation of the system-level data-interchange software of all the test cases presented in Chapter 5 and resulted in simpler system interoperability.

6.3 Limitations

The following sections present limitations of the proposed approach to the automation of the system-level data-interchange software. The limitations cover both the modeling and code generation workflows as proposed by the two-stage workflow presented in Chapter 3. Moreover, these limitations, some of which were identified in preceding chapters, principally originate from the fact that they were not investigated by this research which future work could address.

6.3.1 More than Semantic

SIDL only covers up to the *Semantic* level of the Levels of Conceptual Interoperability Model (LCIM) as defined by [79]. Thus, the following levels of the LCIM are covered by SIDL:

- *Technical* (communication infrastructure)
- *Syntactic* (message format)
- *Semantic* (reference model)

The *Technical* level is covered in SIDL by the *Transport* elements except for binding which is at the *Syntactic* level being the mechanism to format data. The *Syntactic* level also includes all of the *Data* elements except for facts and measures which are at the *Semantic* level providing strong

meaning to data. The other SIDL elements are considered at the *Semantic* level as they encompass a reference model.

The current state of SIDL does not cover the following levels of the LCIM which future work could address:

- *Pragmatic* (information exchange context)
- *Conceptual* (fully specified model)

Supporting higher LCIM levels would enable further system interoperability and automation. As an example, BOM's patterns of interplay [23] capture mission details making them reach the *Pragmatic* level. SIDL buses could be considered to be at the *Pragmatic* level because they capture the concrete link between systems, except this information is not used by systems which would be required to reach this level.

Furthermore, one limitation of SIDL is the ability to completely represent WSDL [28] and TENA [35] models. Both rely on the concept of operations which are not captured in SIDL. One strategy could be to extend the system element to include operations in addition to ports. Another one could be to create higher-level abstractions similar to WSDL's message exchange patterns. Because operations can always be atomically decomposed into data, whether for requests or responses, exchange patterns could be introduced in SIDL's data model. This would also bring SIDL to the *Pragmatic* level.

6.3.2 Conversion Modeling

As previously mentioned in Section 4.2.1, `view` support was limited in this research. On the language side, data adaption was captured by SIDL's model compiler which was able to recognize if either the types matched or not, i.e., if the referenced type of the `select` clause and the type specified with the `as` construct were equal or not. When both types matched, a warning was issued since adaptation was not required, and when both differed, a warning was issued stipulating that the final type was reverted to the referenced type of the `select` clause since no conversions were available to validate or perform the adaptation. Therefore, views were treated as entities by the code generator.

FACE supports conversions in its data model as long as they are between units or frames [26]. Moreover, affine conversions, in the form $mx + b$, can be defined, along with descriptive ones which are meant to be informative and manually implemented by model users.

In order to provide the flexibility required by systems to enable their reuse across multiple platforms in support of product lines, the implementation of views must cover conversions. To this end, SIDL's data model could be extended to incorporate conversions which was preliminary prototyped. The following SIDL code could represent this type of support with affine conversions along with general purpose conversions which would be defined externally and which code generators would have to support.

```
external_conversion GeodeticToGeocentric of Geodetic, Geocentric

affine_conversion DegreesToFahrenheit of Degree, Fahrenheit:
  m: 9/5
  b: 32
```

6.3.3 Defining External Bus Connections

A bus, in SIDL, captures system connection information by having system ports connected to its channels. One limitation of this pattern is the requirement that all systems must connect to the bus at the same time and within the same SIDL description. Moreover, reusing a bus with different connection information is not possible as it stands and would require the introduction of an external connection mechanism. This was prototyped in this research, but unfortunately could not be completed. The following SIDL example demonstrates how the `RadarDisplay` system presented in Chapter 3 could be connected to the `RadarSystemBus` using a new **`connection`** element.

```
connection RadarDisplayConnection of RadarDisplay:
  connect State in RadarSystemBus.RadarState
  connect Detections in RadarSystemBus.Detections
```

6.3.4 Configuration in Support of Modeling

It is interesting to note that the elements captured by SIDL descriptions represent the minimal set of information required to describe system interfaces and their data exchanges. Many information items are not captured by SIDL and are left as configuration data. For instance, the configuration of the DDS and HLA runtime libraries, such as multi-cast groups, are not captured but are

essential in achieving system interoperability. Moreover, middleware configuration data is often vendor-specific. This makes their integration in SIDL requiring further standardization.

The rule to determine if an information item should be captured in SIDL or be left out as configuration data is far from being trivial. Software configuration is so fundamental that FACE proposes standard configuration services [26]. A simplified rule could be to derive new SIDL elements from information items only when they are standardized and impact system interoperability in a uniform way. Nonetheless, this requires further investigation.

6.3.5 Standard SIDL Library Bindings and Metadata Interface

SIDL libraries have been implemented in .NET. It would be interesting to revisit this to determine if it is the proper interchange format or if other platform bindings should be specified instead such as a Java binding. This also requires standardizing the content of the *SIDL.Compiler* library in order to expose a uniform interface to code generators.

6.3.6 Protocol Extensibility

Another area which would benefit system interoperability is through the standardization of SIDL protocol extensions. This is not a trivial task as it touches the core grammar and the interface used to define the protocols. One way of addressing protocol extensibility would be to introduce a protocol element in SIDL which would only capture its characteristics, as opposed to modeling the protocol itself, such as its allowed QoS attributes.

All these limitations highlight the potential growth that comes out of the findings of this thesis and SIDL in particular. Nevertheless, the contributions of this thesis provide a significant step towards simpler system integration and interoperability laying a path ahead filled with even further simplifications.

CONCLUSION

This thesis has addressed the general problem of simplifying system integration and interoperability by automating the system-level data-interchange software through a system interface description language. The current state of the art explored through a comprehensive survey in Chapter 1 shed light on limitations of existing solutions particularly in capturing the system interfaces and considering multi-architecture environments for which no solution existed prior to this work. This literature review also uncovered potential starting points such as using domain-specific languages and code generation techniques. From there, the general methodology of this research was formulated in Chapter 2 with the main research objectives tackling a language for describing system interfaces and the various aspects surrounding their data exchanges, as well as a method for automating the data-interchange software of systems from models described in this language.

SIDL, the System Interface Description Language, introduced in Chapter 3, features the relevant language elements for capturing system interfaces. It covers all the system interoperability facets notably the system *Interface*s, the *Connection* of these interfaces to data, the *Data* exchanged between systems, and the data's *Transport* from system to system. The automation of the software responsible for system data exchanges was achieved through a two-stage workflow involving modeling and code generation stages. This was described in Chapter 4 along with the experimental implementations developed covering the language, the model compiler, the code generator, and their validation. SMEs involved in the system development and integration activities would use the two-stage workflow to create test applications which consisted in the experimental results captured in Chapter 5. The contributions made by this thesis are discussed in detail in Chapter 6 along with limitations which highlight future research directions.

The following sections provide a summary of the contributions made by this thesis along with a vision of future challenges.

Contributions

This thesis has contributed to the simplification of the tasks involved in system integration and interoperability with the goal of reducing their associated costs and increasing their effectiveness along with their efficiency. Even though the primary context of this thesis revolved around full

mission simulators interacting within distributed simulations, the problem of formally describing system interfaces spans more than this context and could apply to other ones such as operational systems.

This thesis contributed a system interface description language (SIDL) and a method for automating the system-level data-interchange software using this language. SIDL is an architecture description language used to formally describe system interfaces focusing on the data they exchange and on the various aspects surrounding them. As a meta-model, it facilitates system interoperability and enables further automation of the tasks involved in achieving it particularly through code generation. Being architecture-agnostic, it provides a single architectural viewpoint overseeing all system interfaces and capturing multi-architecture considerations. The advent of SIDL also contributed a new taxonomy providing a comprehensive perspective over system interoperability.

As a DSL, SIDL provides the richness and expressiveness of a dedicated language to describe system interfaces. The main values of SIDL reside in the easier validation, evolution, and governance of system interfaces it enables. This originates from the human-understandability of the language reflecting the domain of its stakeholders. At the same time, this hides software complexity from them preventing the conflicting duality of requiring them to be experts both in their domain and in areas that are in support of their work such as software development and hardware resource usage. Moreover, acknowledging the current hardware and software trends emphasizes this need even more. The experiences with these new contributions demonstrated concrete gains in these directions. Future work can only improve upon this as the way ahead seems filled with DSLs.

Future Challenges

This section discusses potential ways forward providing a vision over software application development and in improving SIDL as well as enabling system interoperability even further.

Workflow-Driven Development

As highlighted previously, hiding software complexity from SMEs is quite challenging, but the results obtained demonstrate that it is imperative to produce more cost-effective, productive and interoperable software products. Moreover, there is a need to consider the software application as

a whole addressing its cross-cutting concerns (Figure 4) which further increases the challenge. That is why completely hiding software complexity from SMEs following the full product lifecycle management workflow is one future research area exhibiting many challenges.

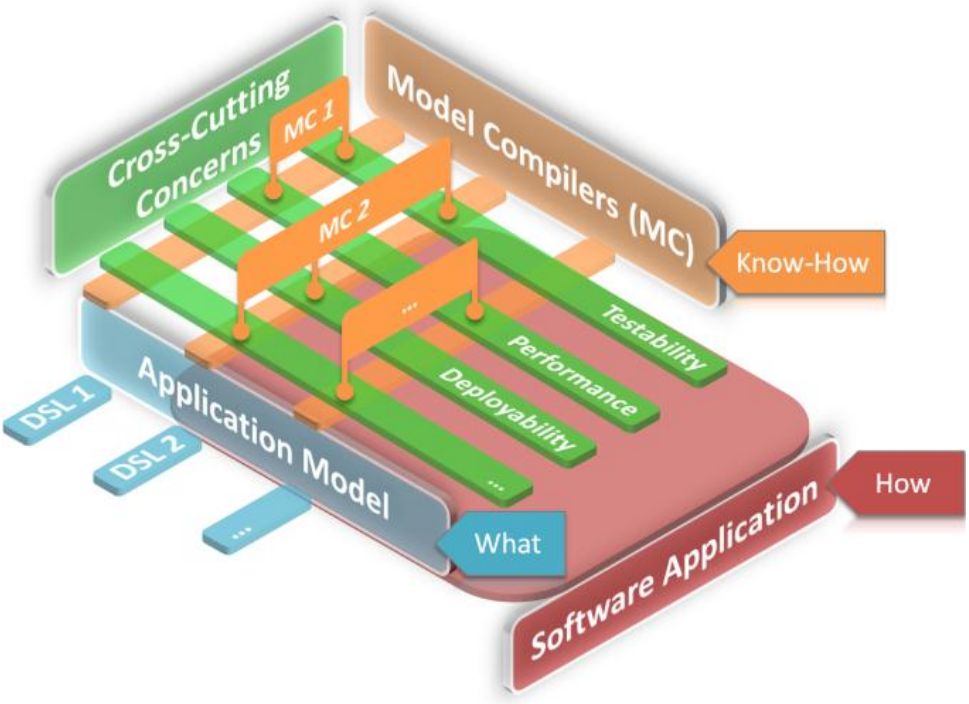

Figure 4: Addressing Cross-Cutting Concerns

Multi-DSLs

As an architecture description language, SIDL focuses on system integration and interoperability. This aspect is one of many others required to fully describe a software application. Since a software application deals with multiple domains and a DSL focuses on one particular domain, multiple DSLs need to be combined to define a complete software application. Otherwise stated, a DSL provides a means to edit a particular aspect, or architectural viewpoint of an application's model (Figure 4). For instance, one could use a graphical DSL implementing a domain specific notation for a coarse-grained cross-cutting concern, like a graphical representation of the computing complex to address the deployment of the software application, while a textual DSL would be used to address a finer-grained cross-cutting concern (Figure 4) such as the mathematical equations involved in the extrapolation of network data to address performance.

Several issues are associated with managing multiple DSLs ranging from mixing graphical and textual DSLs in a single model to integrating multiple DSLs while keeping them as loosely coupled as possible [54]. To this end, Völter [54] highly recommends managing dependencies between DSLs such that there is a strict layering with unidirectional dependencies. For instance, Völter [54] proposes to integrate multiple DSLs by first defining a base DSL on top of an existing framework to simplify its use by raising its level of abstraction and then building a more business-domain specific DSL on top of the base DSL. Also, Johansen [53] underlines that frameworks composition is typically easier because frameworks all use a general purpose language that acts as a common ground. For DSLs to be composable means this common ground needs to be designed up-front. For Fowler and Parsons [80], this common ground is the *Application Model* (Figure 4) (also referred to as the semantic model).

Model Compilers & Legacy Assets

Again, according to the compiler analogy, no one would change assembly code generated by a compiler to modify the behavior of a C++ program. Similarly, no one should think of changing the C++ code generated by a *Model Compiler* to modify the behavior of a DSL program. However, this dogmatic approach collides with reality because in practice not all the software application is auto-generated, such as legacy frameworks and third-party systems. In fact, integrating legacy code requires manual coding and debugging against the auto-generated code. As pointed out by Völter [54], this increases the importance for a *Model Compiler* to generate documented code having well defined extension points for user code. Another source of difficulty is the non-uniformity of some existing legacy APIs, forcing a *Model Compiler* to deviate from its highly uniform model transformations.

Debugging at the DSL Level

The ability for SMEs to debug software applications using DSL concepts is also subject to several issues [54]. Again, a software application is also composed of code that isn't generated by a *Model Compiler*. In particular, since an application such as a FMS is deployed on several hosts, a DSL debugger is required to deal with distributed data. Moreover, some applications are subject to strict performance requirements forcing a *Model Compiler* to support the equivalent of *Debug*

and *Release* builds to avoid the overhead of the additional code needed to enable debugging at the DSL level.

Towards Hardware-Aware Software

The *Application Model* (Figure 4) captures several cross-cutting concerns and model-specific details giving *Model Compiler*s the necessary context and leeway to make better decisions regarding hardware optimizations. Even though the software application has strong hardware dependencies, it is the *Model Compiler*s that provide cross-platform portability by adapting the *Application Model* towards a specific platform. However, moving towards hardware-aware software to obtain better performance requires a *Know-How* (Figure 1-4) of a hardware specific set of changes to the *Application Model* that increases the gap between the DSL concepts and the structure of the associated code. Fortunately, as pointed out by Völter [54], integrating multiple DSLs via several *Model Compiler*s as part of a transformation chain seems applicable to deal with this additional complexity.

Modeling Solution

DSLs are designed with the goal of implementing a solution to a specific problem domain. To be successful, one also needs to consider a particular domain workflow. To this end, the more we hide software complexity from SMEs, the more we are able to identify and formalize the fundamental tasks of SMEs which define their workflow. In the end, we are still creating software that is an economic asset and that needs to deliver the required performances while being maintainable. This asset, the software application, still needs to be updated with patches and service packs. Some software quality attributes cannot be hidden from the SMEs because ultimately they are dealing with software. A major challenge is to provide a productive infrastructure by maintaining the level of abstraction across DSLs with tools supporting the SMEs' workflow which encompasses the full product lifecycle.

Capturing Data Model Mappings

Early work has started to extend SIDL in order to capture the mapping between data models. The end goal is to simplify gateway creation by providing an architecture-agnostic way of specifying the mappings. One of the test applications developed was a DDS-DIS gateway to bridge two

SIDL networks. It reused the same code generation as their individual DDS and DIS application counterparts. Preliminary work started to capture these mappings in order to simplify and further automate this application. This is a natural evolution of SIDL and is in line with system interface adaptation in order to simplify system integration further.

BIBLIOGRAPHY

[1] Robert Lutz et al., "A Systems Engineering Perspective on the Development and Execution of Multi-Architecture LVC Environments," in *Simulation Interoperability Workshop*, Orlando, FL, 2010.

[2] Jeffrey Wallace et al., "Object Model Composability and LVC interoperability Update," in *Fall Simulation Interoperability Workshop*, 2009.

[3] "Systems and software engineering - Architecture description," ISO/IEC/IEEE, ISO/IEC/IEEE 42010:2011, 2011.

[4] Duncan C Miller and Jack A Thorpe, "SIMNET: The advent of simulator networking," *Proceedings of the IEEE*, vol. 83, no. 8, pp. 1114-1123, 1995.

[5] L. N. Cosby, "SIMNET: An Insider's Perspective," IDA-D-1661, 1995. [Online]. https://www.dtic.mil/dtic/tr/fulltext/u2/a294786.pdf

[6] "IEEE Standard for Distributed Interactive Simulation-Application Protocols," IEEE, IEEE Std 1278.1-2012, 2012.

[7] Ronald C Hofer and Margaret L Loper, "DIS today [Distributed interactive simulation]," *Proceedings of the IEEE*, vol. 83, no. 8, pp. 1124-1137, 1995.

[8] J. S. Gansler. (1998) High Level Architecture Module 2 Advanced Topics. [Online]. http://www.ecst.csuchico.edu/~hla/LectureNotes/Policy.pdf

[9] Judith S Dahmann and Katherine L Morse, "High level architecture for simulation: An update," in *Distributed Interactive Simulation and Real-Time Applications, 1998. Proceedings. 2nd International Workshop on*, 1998, pp. 32-40.

[10] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules," IEEE, IEEE Std 1516-2010, 2010.

[11] Katherine L Morse, Lubomir Bic, Michael Dillencourt, and Kevin Tsai, "Multicast grouping for dynamic data distribution management," in *Summer Computer Simulation Conference*, 1999, pp. 312-318.

[12] Katherine L Morse, Lubomir Bic, and Michael Dillencourt, "Interest management in large-scale virtual environments," *Presence: Teleoperators & Virtual Environments*, vol. 9, no. 1, pp. 52-68, 2000.

[13] K. L. Morse and M. D. Petty, "Data distribution management migration from DoD 1.3 to IEEE 1516," in *Proceedings Fifth International Workshop on Distributed Simulation and Real-Time Applications*, 2001, pp. 58-65.

[14] Lu Tanchi, Lee Chungnan, Hsia Wenyang, and Lin Mingtang, "Supporting large-scale distributed simulation using HLA," *ACM Transactions on Modeling and Computer Simulation*, vol. 10, no. 3, pp. 268-294, 2000.

[15] David S Dodge, "Gateways, A Necessary Evil?," in *Simulation Interoperability Standards Organization (SISO) Fall Simulation Interoperability Workshop*, 2000.

[16] David S Dodge, "Gateways-101," in *Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force. IEEE*, vol. 1, 2001, pp. 532-538.

[17] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) Object Model Template (OMT) Specification," IEEE, IEEE Std 1516.2-2010, 2010.

[18] Wesley K. Braudaway and Susan M. Harkrider, "Implementation of the High Level Architecture into DIS-Based Legacy Simulations," in *Simulation Interoperability Workshop*, 1997.

[19] Hakan Savaşan, İldeniz Duman, Mustafa Dinç, and İlker Şahin, "Migrating a Legacy Simulation to HLA: Lessons Learned Integrating with New Native HLA Simulations," in

Simulation Interoperability Workshop, 2003.

[20] Björn Möller and Lennart Olsson, "Practical experiences from HLA 1.3 to HLA IEEE 1516 Interoperability," *Simulation Interoperability Standards Organization (SISO)*, 2004.

[21] A. Cox, D. Wood, M. Petty, and K. Juge, "Integrating DIS and SIMNET into HLA with a gateway," in *DIS Workshop*, 1996.

[22] "IEEE Recommended Practice for Distributed Simulation Engineering and Execution Process (DSEEP)," SISO, IEEE Std 1730-2010, 2011.

[23] "Standard for Base Object Model (BOM) Template Specification," SISO, SISO-STD-003-2006, 2006.

[24] Xu Xie, Xiaocheng Liu, Ying Cai, and Kedi Huang, "Research on SDEM and Its Transformation in the Gateway Design," in *AsiaSim 2012*.: Springer, 2012, pp. 222-230.

[25] Michael J O'Connor, K Lessmann, and DL Drake, "LVCAR Enhancements for Using Gateways," in *Spring Simulation Interoperability Workshop*, 2011.

[26] "Technical Standard for Future Airborne Capability Environment (FACE™), Edition 2.0," The Open Group, C137, 2013. [Online]. https://www2.opengroup.org/ogsys/catalog/c137

[27] M. Henry et al., "A comparison of open architecture standards for the development of complex military systems: GRA, FACE, SCA NeXT (4.0)," in *Military Communications Conference, 2012 - MILCOM 2012*, 2012, pp. 1-9.

[28] "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," W3C, wsdl20, 2007. [Online]. http://www.w3.org/TR/wsdl20

[29] "Service-Oriented Architecture Ontology," The Open Group, C104, 2010.

[30] "XML Schema Part 2: Datatypes Second Edition," W3C, xmlschema-2, 2004. [Online]. http://www.w3.org/TR/xmlschema-2

[31] Joe McKendrick. (2013, July) 10 steps to avoid cloud vendor lock-in. [Online]. http://www.zdnet.com/10-steps-to-avoid-cloud-vendor-lock-in-7000017971

[32] "Interface Definition Language 3.5," Object Management Group, 2013.

[33] "C++ Language Mapping," Object Management Group, formal/2012-07-02, 2012.

[34] "Data Distribution Service for Real-time Systems Version 1.2," Object Management Group, formal/07-01-01, 2007.

[35] J. Russell Noseworthy. (2004) The TENA Middleware - Model-Based Distributed Application Development for High-Reliability DoD Range Systems. [Online]. http://www.omg.org/news/meetings/workshops/RT_2004_Manual/10-3_Noseworthy.pdf

[36] Stacy VanWinkle. (2011) TENA Release 6.0.1 Technical Introduction Course (TIC). [Online]. https://www.tena-sda.org/download/attachments/6750/TENA-v6.0.1-TIC-2011-06-16.pdf

[37] J. Russell Noseworthy, "Developing distributed applications rapidly and reliably using the TENA middleware," in *Military Communications Conference, 2005. MILCOM 2005. IEEE*, vol. 3, Atlantic City, NJ, 2005, pp. 1507-1513.

[38] "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation," International Telecommunication Union, ISO/IEC 8824-1, 2008.

[39] Olivier Dubuisson, *ASN. 1: communication between heterogeneous systems*.: Morgan Kaufmann, 2001.

[40] Peter H Feiler and David P Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*.: Addison-Wesley Professional, 2012.

[41] "OMG Systems Modeling Language (OMG SysML) Version 1.3," Object Management

Group, formal/2012-06-01, 2012.

[42] Moritz Eysholdt and Johannes Rupprecht, "Migrating a large modeling environment from XML/UML to Xtext/GMF," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2010, pp. 97-104.

[43] "XML Metadata Interchange Specification Version 2.0.1," Object Management Group, ISO/IEC 19503:2005, 2005.

[44] Holger Eichelberger, Yilmaz Eldogan, and Klaus Schmid, "A Comprehensive Survey of UML Compliance in Current Modelling Tools," *Software Engineering*, vol. 143, pp. 39-50, 2009.

[45] Tom Morris. (2008) What's wrong with UML?. [Online]. http://tfmorris.blogspot.ca/2008/10/whats-wrong-with-uml.html

[46] George Mamais, Thanassis Tsiodras, David Lesens, and Maxime Perrotin, "An ASN.1 compiler1 for embedded/space systems," in *ERTS*, Toulouse, France, 2012.

[47] Thomas Vergnaud, Jerome Hugues, Laurent Pautet, and Fabrice Kordon, "Rapid development methodology for customized middleware," in *Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP 2005)*, Montreal, Que., Canada, 2005, pp. 111-117.

[48] Martin Tapp, Sidney Chartrand, and Jean-François Campeau, "Experiences in Leveraging M&S Expertise by Hiding Software Complexity," in *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*, Orlando, 2011.

[49] Robert B France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg, "Model-driven development using UML 2.0: promises and pitfalls," *Computer*, vol. 39, no. 2, pp. 59-66, 2006.

[50] Andrew Brownsword. (2007) A Game Developer's Perspective On Parallelism. [Online]. http://research.microsoft.com/apps/video/default.aspx?id=103943

[51] Daniel Collin. (2010) DICE Publications. [Online]. http://publications.dice.se/attachments/Introduction_to_Data-Oriented_Design.pdf

[52] Intel. (2011) Intel® 64 and IA-32 Architectures Optimization Reference Manual. [Online]. http://www.intel.com/Assets/PDF/manual/248966.pdf

[53] Martin Fagereng Johansen, "Domain Specific Languages versus Frameworks," University of Oslo, Oslo, Master Thesis 2009. [Online]. http://folk.uio.no/martifag/papers/Johansen2009.pdf

[54] Markus Völter. (2011) MD*/DSL Best Practices. [Online]. http://www.voelter.de/data/pub/DSLBestPractices-2011Update.pdf

[55] Debasish Ghosh, *DSLs in Action*.: Manning Publications, 2010.

[56] Wenguang Wang, Andreas Tolk, and Weiping Wang, "The levels of conceptual interoperability model: Applying systems engineering principles to M&S," in *Proceedings of the 2009 Spring Simulation Multiconference*, San Diego, 2009.

[57] J. Russell Noseworthy. (2003) The TENA Middleware - Supporting Real-Time Application Development for the DoD Range Community. [Online]. http://www.omg.org/news/meetings/workshops/RT_2003_Manual/Presentations/2-2_Noseworthy.pdf

[58] "IEEE Standard for Configuration Management in Systems and Software Engineering," IEEE Computer Society, IEEE Std 828-2012, 2012.

[59] César de la Torre Llorente, "Model-Driven SOA with Oslo," *The Architecture Journal*, vol. 21, pp. 10-15, 2009.

[60] Martin Tapp, Brian Pages, Gabriela Nicolescu, and El Mostapha Abouhamid, "A Generalized Approach to Networked Systems Interoperability," in *Simulation Interoperability Workshop*, Orlando, 2005, pp. 507-518.

[61] Martin Tapp, Gabriela Nicolescu, and El Mostapha Aboulhamid, "Experiences with an XML Format & Syntax for Describing Interoperability," in *Simulation Interoperability Workshop*, vol. 2, Huntsville, 2006, pp. 601-612.

[62] Martin Tapp, Gabriela Nicolescu, and El Mostapha Aboulhamid, "A Performance Evaluation of Dynamically Generated Gateways," in *Simulation Interoperability Workshop*, 2006.

[63] Martin Tapp, "System Interface Description Language," in *Simulation interoperability Workshop*, Orlando, 2013.

[64] "Real-time Platform Reference Federation Object Model," SISO, SISO-STD-001.1-1999, 1999.

[65] "Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane," ISO, ISO/IEC 10646-1, 2000.

[66] "C# Language Specification 4th Edition," ECMA International, ISO/IEC 23270:2006, 2006.

[67] rollynoel. (2013) Boo Primer: [Part 02] Variables - List of Value Types. [Online]. https://github.com/bamboo/boo/wiki/Boo-Primer:-%5BPart-02%5D-Variables#wiki-ListOfValueTypes

[68] Peter H. Feiler, David P. Gluch, and John J. Hudak, "The Architecture Analysis & Design Language (AADL): An Introduction," Carnegie-Mellon University of Pittsburgh PA Software Engineering, CMU/SEI-2006-TN-011, 2006.

[69] IC#Code. (2012) The Open Source Development Environment for.NET. [Online].

http://www.icsharpcode.net/OpenSource/SD/

[70] Codehaus. (2009) Boo A wrist friendly language for the CLI. [Online]. http://boo.codehaus.org/

[71] "Common Language Infrastructure (CLI) 6th Edition," ECMA International, ISO/IEC 23271, 2012.

[72] Ayende Rahien, *DSLs in Boo: Domain Specific Languages in. NET*.: Manning Publications Co., 2010.

[73] The Eclipse Foundation. (2013) Eclipse Modeling Framework Project (EMF). [Online]. http://www.eclipse.org/modeling/emf/

[74] Martin Fowler. (2005) Language Workbenches: The Killer-App for Domain Specific Languages? [Online]. http://martinfowler.com/articles/languageWorkbench.html

[75] Microsoft Inc. (2013) Strong-Named Assemblies. [Online]. http://msdn.microsoft.com/en-us/library/wd40t7ad.aspx

[76] Microsoft Inc. (2013) Code Generation and T4 Text Templates. [Online]. http://msdn.microsoft.com/en-us/library/vstudio/bb126445.aspx

[77] Björn Möller, Mikael Karlsson, and Björn Löfstrand, "Reducing Integration Time and Risk with the HLA Evolved Encoding Helpers," in *2006 Spring Simulation Interoperability Workshop*, 2006.

[78] Jim Webber, Savas Parastatidis, and Ian Robinson, *REST in Practice: Hypermedia and Systems Architecture*.: O'Reilly Media, Inc., 2010.

[79] Andreas Tolk, Charles Turnitsa, and Saikou Diallo, "Implied ontological representation within the levels of conceptual interoperability model," *Intelligent Decision Technologies*, vol. 2, no. 1, pp. 3-19, February 2008.

[80] Martin Fowler and Rebecca Parsons, *Domain-Specific Languages*.: Addison-Wesley Professional, 2010.

[81] W3C. (2004) Extensible Markup Language (XML) - Notation. [Online]. http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation

[82] Paul Young, Nabendu Chaki, and Valdis Berzins, "Evaluation of middleware architectures in achieving system interoperability," in *Proceedings 14th IEEE International Workshop on Rapid Systems Prototyping*, San Diego, CA, 2003, pp. 108-116.

[83] Tobias Walter and Jürgen Ebert, "Combining DSLs and ontologies using metamodel integration," in *Domain-Specific Languages*, 2009, pp. 148-169.

[84] Tobias Walter, "Combining Domain-Specific Languages and Ontology Technologies," *Technical Report 2009-566 School of Computing, Queen's University Kingston, Ontario, Canada*, p. 34, 2009.

[85] Markus Völter. (2011) Type Systems for DSLs. [Online]. http://www.infoq.com/presentations/Type-Systems-for-DSLs

[86] L. van Ruijven, "Ontology for Systems Engineering: Model-Based Systems Engineering," in *Computer Modeling and Simulation (EMS), 2012 Sixth UKSim/AMSS European Symposium on*, 2012, pp. 371-376.

[87] Axel Uhl, "Model-driven development in the enterprise," *IEEE Software*, vol. 25, no. 1, pp. 46-49, 2008.

[88] Andreas Tolk, Saikou Y. Diallo, and Charles D. Turnitsa, "Ontology Driven Interoperability – M&S Applications," in *I/ITSEC*, Suffolk, 2006.

[89] Vincent Thomson and Mohammad Raffay Zaidi, "Product Interface Management: Methods for effective modelling of product subsystem interface," McGill University, Master Thesis 2011.

[90] Vincent Thomson and Sofiene Boujbel, "Product Interface Management: Developing a standard ontology to describe product subsystem interfaces," 2010.

[91] Martin Tapp, Michel Patenaude, and Jan Prawdzik, "Instructor Station Challenges Of Monitoring Distributed Simulations: An XML Solution?," in *Simulation Technology and Training Conference*, Melbourne, 2002.

[92] G. Tan, A. Persson, and R. Ayani, "HLA federate migration," in *Proceedings. 38th Annual Simulation Symposium*, San Diego, CA, 2005, pp. 243-50.

[93] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro, *EMF: eclipse modeling framework*.: Addison-Wesley Professional, 2008.

[94] Thomas Stahl and Markus Voelter, *Model-driven software development*.: John Wiley & Sons Chichester, 2006.

[95] Douglas C. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25-31, 2006.

[96] André L Santos, Kai Koskimies, and Antónia Lopes, "Automated domain-specific modeling languages for generating framework-based applications," in *Software Product Line Conference, 2008. SPLC'08. 12th International*, 2008, pp. 149-158.

[97] Igor Sacevski and Jadranka Veseli, "Introduction to Model Driven Architecture (MDA)," in *Seminar Paper, University of Salzburg*, 2007.

[98] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*.: Pearson Higher Education, 2004.

[99] Keyvan Rahmani and Vincent Thomson, "Ontology based interface design and control methodology for collaborative product development," *Computer-Aided Design*, vol. 44, no. 5, pp. 432-444, 2012.

[100] Thomas Quinot, Fabrice Kordon, and Laurent Pautet, "From functional to architectural analysis of a middleware supporting interoperability across heterogeneous distribution models," in *Distributed Objects and Applications, 2001. DOA'01. Proceedings. 3rd International Symposium on*, Rome, Italy, 2001, pp. 165-175.

[101] T. Quinot, F. Kordon, and L. Pautet, "Architecture for a reuseable object-oriented polymorphic middleware," in *Pdpta'2001: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, 2001, pp. 1994-2001.

[102] Daniel J Paterson, Eric Anschuetz, Mark Biddle, and Dave Kotick, "An Approach to HLA Gateway/Middleware Development," in *Simulation Interoperability Workshop*, 1998.

[103] Daniel J Paterson, Erik S Hougland, and Juan J Sanmiguel, "A Gateway/Middleware HLA implementation and the extra Services that can be provided to the Simulation," in *Fall Simulation Interoperability Workshop*, 2000.

[104] P. E. Papotti, A. F. do Prado, and W. L. de Souza, "An approach to support legacy systems reengineering to MDD using metaprogramming," in *Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En*, 2012, pp. 1-10.

[105] Andreas L. Opdahl, "A Platform for Interoperable Domain-Specific Enterprise Modelling Based on ISO 15926," in *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2010 14th IEEE International*, 2010, pp. 301-310.

[106] K. L. Morse and M. Zyda, "Multicast grouping for data distribution management," *Simulation Practice and Theory*, vol. 9, no. 3-5, pp. 121-41, 2002.

[107] K. L. Morse and M. D. Petty, "High Level Architecture data distribution management migration from DoD 1.3 to IEEE 1516," *Concurrency and Computation Practice and Experience*, vol. 16, no. 15, pp. 1527-43, 2004.

[108] K. L. Morse, M. Lightner, R. Little, B. Lutz, and R. Scrudder, "Enabling simulation interoperability," *Computer*, vol. 39, no. 1, pp. 115-17, 2006.

[109] K. L. Morse, "Data and metadata requirements for composable mission space environments," in *Proceedings of the 2004 Winter Simulation Conference*, Washington, DC, 2004, pp. 271-8.

[110] Thom McLean, Richard Fujimoto, and Brad Fitzgibbons, "Middleware for real-time distributed simulations," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 15, pp. 1483-1501, 2004.

[111] Leon McGinnis and Volkan Ustun, "A simple example of SysML-driven simulation," in *Simulation Conference (WSC), Proceedings of the 2009 Winter*, 2009, pp. 1703-1710.

[112] Sjouke Mauw, Wouter T Wiersma, and Tim AC Willemse, "Language-driven system design," in *System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on*, 2002, pp. 3637-3646.

[113] Anders Mattsson, Björn Lundell, Brian Lings, and Brian Fitzgerald, "Linking model-driven development and software architecture: A case study," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 83-93, 2009.

[114] Sonja Maier and Daniel Volk, "Facilitating language-oriented game development by the help of language workbenches," in *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, Toronto, Ontario, Canada, 2008, pp. 224-227.

[115] James Lapalme et al.,".NET Framework - A Solution for the Next Generation Tools for System-Level Modeling and Simulation," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 1, 2004, pp. 732-733.

[116] Steven Kelly. (2012) Concrete Syntax Matters. [Online]. http://www.infoq.com/presentations/Language-Design

[117] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev, "TCS: a DSL for the specification of textual concrete syntaxes in model engineering," in *Proceedings of the 5th international conference on Generative programming and component engineering*, 2006, pp. 249-254.

[118] Andreas Jordan, Georg Grossmann, Wolfgang Mayer, Matt Selway, and Markus Stumptner, "On the application of software modelling principles on ISO 15926," in *Proceedings of the Modelling of the Physical World Workshop*, New York, NY, USA, 2012, p. 3.

[119] Werner Heijstek and Michel RV Chaudron, "The impact of model driven development on the software architecture process," in *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, 2010, pp. 333-341.

[120] Paul E Hanover, "Transitioning from DIS to HLA in a Distributed Simulation Environment," in *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*, vol. 2005, 2005.

[121] Richard C Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*.: Addison-Wesley Professional, 2009.

[122] Ian Gorton and Yan Liu, "Advancing software architecture modeling for large scale heterogeneous systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, Santa Fe, New Mexico, USA, 2010, pp. 143-148.

[123] G. Giachetti, F. Valverde, and B. Marin, "Interoperability for model-driven development: Current state and future challenges," in *Research Challenges in Information Science (RCIS), 2012 Sixth International Conference on*, 2012, pp. 1-10.

[124] Bingfeng Ge, Keith W Hipel, Long Li, and Yingwu Chen, "A data-centric executable modeling approach for system-of-systems architecture," in *System of Systems Engineering (SoSE), 2012 7th International Conference on*, 2012, pp. 368-373.

[125] T. Gaska, "Optimizing an incremental Modular Open System Approach (MOSA) in avionics systems for balanced architecture decisions," in *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, 2012, pp. 7D1-1-7D1-19.

[126] Martin Fowler. (2004) ModelDrivenArchitecture. [Online]. http://www.martinfowler.com/bliki/ModelDrivenArchitecture.html

[127] Martin Fowler. (2005) Language Workbenches and Model Driven Architecture. [Online]. http://www.martinfowler.com/articles/mdaLanguageWorkbench.html

[128] Roy T. Fielding. (2008) REST APIs must be hypertext-driven. [Online]. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[129] Roy Thomas Fielding. (2000) Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation.

[130] Peter H Feiler, Jorgen Hansson, Dionisio De Niz, and Lutz Wrage, "System architecture virtual integration: An industrial case study," DTIC Document, 2009.

[131] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*.: Addison-Wesley, 2003.

[132] Thomas A DuBois et al., "Open Networking Technologies for the Integration of Net-Ready Applications on Rotorcraft," in *Annual conference of the American Helicopter Society*, 2012.

[133] J. Dingel, D. Garlan, and C. Damon, "Bridging the HLA: problems and solutions," in *Proceedings Sixth IEEE International Workshop on Distributed Simulation and Real-Time Applications*, Fort Worth, TX, 2002, pp. 33-42.

[134] Jesús Sánchez Cuadrado and Jesús García Molina, "Building domain-specific languages for model-driven development," *IEEE Software*, vol. 24, no. 5, pp. 48-55, 2007.

[135] Jak Charlton. (2009) DDD: The Ubiquitous Language. [Online]. http://devlicio.us/blogs/casey/archive/2009/02/09/ddd-the-ubiquitous-language.aspx

[136] Ines Čeh, Matej Črepinšek, Tomaž Kosar, and Marjan Mernik, "Using ontology in the development of domain-specific languages," *INFORUM 2010*, pp. 185-196, 2010.

[137] Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener, "Efficient Wire Formats for High Performance Computing," in *Supercomputing, ACM/IEEE 2000 Conference*, 2000, pp. 39-39.

[138] Andrew J. Brust. (2010) So Long Oslo, We Hardly Knew Ye. [Online]. http://visualstudiomagazine.com/blogs/redmond-review/2010/09/death-of-oslo.aspx

[139] Benoît Bréholée and Pierre Siron, "Design and Implementation of a HLA Inter-Federation Bridge," in *Proceedings of the 2003 Euro Simulation Interoperability Workshop*, Stockholm, Sweden, 2003.

[140] Zohra Boudjemil, Patrick Phelan, Miguel Ponce de Leon, and Sven van der Meer, "A Case Study for Defining Interoperable Network Components Using MDD," in *Computer Modeling and Simulation (EMS), 2010 Fourth UKSim European Symposium on*, 2010, pp. 381-386.

[141] Jason Bloomberg, *The Agile Architecture Revolution: How Cloud Computing, REST-based SOA, and Mobile Computing are Changing Enterprise IT*.: Wiley, 2013.

[142] Michael Bleigh. (2010) REST isn't what you think it is, and that's OK. [Online]. http://intridea.com/2010/4/29/rest-isnt-what-you-think-it-is?blog=company

[143] Ge Bingfeng, K. W. Hipel, Li Long, and Chen Yingwu, "A data-centric executable modeling approach for system-of-systems architecture," in *System of Systems Engineering (SoSE), 2012 7th International Conference on*, 2012, pp. 368-373.

[144] Ashwini Barshikar, "Communication code generation for distributed applications," M.c.s.

2001.

[145] Peter Barna, Flavius Frasincar, and Geert-Jan Houben, "A workflow-driven design of web information systems," in *Proceedings of the 6th international conference on Web engineering*, 2006, pp. 321-328.

[146] Ritu Arora and Purushotham Bangalore, "A framework for raising the level of abstraction of explicit parallelization," in *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, 2009, pp. 339-342.

[147] Margarida Afonso, Regis Vogel, and Jose Teixeira, "From code centric to model centric software engineering: practical case study of MDD infusion in a systems integration company," in *Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software, 2006. MBD/MOMPES 2006*, 2006, pp. 10 pp.-134.

[148] El Mostapha Aboulhamid and Frédéric Rousseau, *System Level Design with. NET Technology*.: CRC Press, 2009.

[149] "TOGAF Version 9.1," The Open Group, G116, 2011.

[150] "The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification Version 2.1," Object Management Group, formal/2009-01-05, 2009.

[151] DoD Deputy Chief Information Officer. (2010) The DoDAF Architecture Framework Version 2.02. [Online]. http://dodcio.defense.gov/dodaf20.aspx

[152] "Systems and software engineering - System life cycle processes," ISO, ISO/IEC 15288:2008, 2008.

[153] "SOA Reference Architecture," The Open Group, C119, 2011.

[154] "SOA Governance Framework," The Open Group, C093, 2009.

[155] Carnegie Mellon University. (2011) SAE International AS5506A (AADL V2) Syntax Cheat Sheet. [Online]. http://www.aadl.info/aadl/documents/AADL%20V2.1%20Syntax%20Card.pdf

[156] "OMG Unified Modeling Language (OMG UML), Infrastructure," Object Management Group, ISO 19505-1:2012, 2011.

[157] "OMG Object Constraint Language (OCL) Version 2.3.1," Object Management Group, formal/2012-01-01, 2012.

[158] Object Management Group. (2013) OMG Model Driven Architecture. [Online]. http://www.omg.org/mda

[159] "OMG Meta Object Facility (MOF) Core Specification Version 2.4.1," Object Management Group, ISO 19502:2005, 2011.

[160] "Model Driven Message Interoperability," Object Management Group, formal/2010-03-01, 2010.

[161] "Guide for BOM Use and Implementation," SISO, SISO-STD-003.1-2006, 2006.

[162] "Guidance for Design of Aircraft Equipment and Software for Use in Training Devices," ARINC, 610C, 2009.

[163] "ECMAScript Language Specification Edition 5.1," ECMA International, ISO/IEC 16262:2011, 2011.

[164] "Adoption of ISO/IEC 15288:2002 Systems Engineering - System Life Cycle Processes," IEEE Computer Society, IEEE Std 15288-2004, 2004.

[165] Robert J. Allen, "A Formal Approach to Software Architecture," Carnegie-mellon University, Pittsburgh, PA, CMU-CS-97-144, 1997.

[166] Ivano Malavolta, "Software Architecture Modeling by Reuse, Composition and Customization," Universita di L'Aquila, L'Aquila, Italy, Thesis 2012.

[167] Massachusetts Institute of Technology. (2009) Lecture 8 – Systems Integration and Interface Management. [Online]. http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-842-fundamentals-of-systems-engineering-fall-2009/lecture-notes/MIT16_842F09_lec08.pdf

[168] B. Sarder and S. Ferreira, "Developing Systems Engineering Ontologies," in *System of Systems Engineering, 2007. SoSE '07. IEEE International Conference on*, 2007, pp. 1-6.

[169] Kirsten Mewes, "Domain-specific Modelling of Railway Control Systems with Integrated Veri," Universit□at Bremen, Thesis 2010.

[170] Marco Brambilla, Jordi Cabot, and Manuel Wimmer, *Model-Driven Software Engineering in Practice*.: Morgan & Claypool, 2012.

[171] Amine El Kouhen, Cédric Dumoulin, Sébastien Gerard, and Pierre Boulet, "Evaluation of Modeling Tools Adaptation," 2012. [Online]. http://hal.archives-ouvertes.fr/hal-00706701

[172] Keith A. Rigby, "Interface Management," in *Aircraft Systems Integration of Air-Launched Weapons*.: John Wiley & Sons Ltd., 2013, ch. 8, pp. 111-124.

[173] Saikou Y. Diallo and José J. Padilla, "Military Interoperability Challenges," in *Handbook of Real-world Applications in Modeling and Simulation*.: John Wiley & Sons, 2012, vol. 2, ch. 8.

[174] Tobias Walter, Fernando Silva Parreiras, and Steffen Staab, "Ontodsl: An ontology-based framework for domain-specific languages," in *Model Driven Engineering Languages and Systems*. Denver, CO, USA: Springer, 2009, pp. 408-422.

[175] Matthew Stephan and James R. Cordy, "A Survey of Methods and Applications of Model Comparison," School of Computing, Queen's University, Kingston, Ontario, Canada,

Technical Report 2011-582 Rev. 3, 2012.

[176] Michael Rauch and Christoph Gutmann. (2013) Model-driven Development in the Context of Technical SOA. [Online]. http://www.infoq.com/presentations/Model-Driven-SOA

[177] Wikipedia. (2013) List of Unified Modeling Language tools. [Online]. http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

[178] Daryl Ralph Hild, "Discrete event system specification (devs) distributed object computing (doc) modeling and simulation," University of Arizona, Thesis 2000.

[179] "Web Services Architecture," W3C, ws-arch, 2004. [Online]. http://www.w3.org/TR/ws-arch

[180] "Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts," W3C, wsdl20-adjuncts, 2007. [Online]. http://www.w3.org/TR/wsdl20-adjuncts

[181] "Web Services Metadata Exchange (WS-MetadataExchange)," W3C, WS-MetadataExchange, 2011. [Online]. http://www.w3.org/TR/ws-metadata-exchange/

[182] "XML Information Set (Second Edition)," W3C, xml-infoset, 2004. [Online]. http://www.w3.org/TR/xml-infoset

[183] U.S. Department of Defense, "High-Level Architecture Object Model Template Specification Version 1.3," 1998.

[184] D. Fay, "An Architecture for Distributed Applications on the Internet: Overview of Microsoft's.NET Platform," in *International Parallel and Distributed Processing Symposium*, 2003.

[185] Microsoft Corporation. (2004) Microsoft.NET Development Center. [Online]. http://msdn.microsoft.com/netframework

[186] (2004) Mono Project. [Online]. http://www.mono-project.com

[187] OneSAF. (2003) OneSAF Testbed Baseline (OTB). [Online]. http://www.onesaf.org

[188] CAE Inc. (2000) STRIVE. [Online]. http://www.cae.com

[189] Michael H. Lutz and Phillip A. Laplante, "C# and the.NET Framework: Ready for Real Time?," *IEEE Software*, vol. 20, pp. 74-80, 2003.

[190] "XSL Transformations (XSLT)," W3C, xslt, 1999. [Online]. http://www.w3.org/TR/xslt

[191] "XML Path Language (XPath)," W3C, xpath, 1999.

[192] F. I. Vokolos and E. J. Weyuker, "Performance Testing of Software Systems," in *Workshop on Software and Performance*, 1998.

[193] Dan Chen, Bu-Sung Lee, Wentong Cai, and Stephen John Turner, "Design and Development of a Cluster Gateway for Cluster-based HLA Distributed Virtual Simulation Environments," in *Annual Simulation Symposium, Proceedings of the 36th Annual Simulation Symposium*, 2003, p. 193.

[194] J. Latour and M. Adelantado, "Performance Evaluation of an Intuitive C++ Language for the Design of HLA Federates and Federations," in *Simulation Interoperability Workshop*, 2004.

[195] Bradford Dillman, Jason Murphy, and Dan Bleichman, "Effects of high latency wide area networks in distributed simulation," in *Simulation Interoperability Workshop*, 2005.

[196] J. Szulinksi and C. Simpkins, "Latency Testing In the USAF Distributed Mission Operations Environment," in *Simulation Interoperability Workshop*, 2005.

[197] Microsoft Corp. (2005) Platform Invocation Services - PInvoke and the DllImport Attribute. [Online]. http://msdn.microsoft.com/library

[198] "C++/CLI Language Specification," ECMA, Standard 372, 2005.

[199] "Guidance, Rationale, and Interoperability Manual (GRIM) for the Real-time Platform Reference Federation Object Model (RPR FOM)," SISO, SISO-STD-001-1999, 1999.

APPENDIX

APPENDIX A SIDL GRAMMAR REFERENCE

The following provides the SIDL grammar reference. The format used is a simplified version of the Extended Backus–Naur Form (EBNF) for XML [81] with the following distinctions: productions use '=' instead of '::='. The presented grammar only focuses on the language constructs. Therefore, other constructs are possible such as single line comments, which start with '//', and multi-line comments, which are enclosed within '/*' and '*/'. Moreover, control blocks are delimited by whitespace which removes the need for explicit delimiters such as curly braces '{}' in C, C++, and Java. Whitespace includes any character in Unicode class Zs, horizontal tab (U+0009), vertical tab (U+000B), and form feed (U+000C) [66]. Additionally, long lines can be broken-up with the line-continuation character which is the backslash '\'.

```
sidl_description =
  namespace_directive?
  import_directive*
  declaration*

namespace_directive = 'namespace' identifier
import_directive = 'import' identifier ('as' ID)?

declaration =
    data_facet_decl
  | interface_facet_decl
  | connection_facet_decl
  | transport_facet_decl

data_facet_decl =
    observable_decl
  | info_decl
  | unit_decl
  | frame_decl
  | measure_decl
  | fact_decl
  | enum_decl
  | variant_decl
  | entity_decl
  | view_decl

interface_facet_decl =
  system_decl

connection_facet_decl =
    bus_decl

transport_facet_decl =
    binding_decl
  | network_decl

observable_decl = 'observable' ID
info_decl = 'info' ID
unit_decl = 'unit' ID
frame_decl = 'frame' ID

measure_decl =
    simple_measure
  | composite_measure
```

```
simple_measure =
    'measure' ID 'of' observable_ref 'as' value_type (':' simple_measure_property+)?

simple_measure_property =
    'units' unit_ref
  | 'frame' frame_ref
  | 'precision' real_literal

composite_measure = 'measure' ID 'of' observable_ref (':' field_decl+)?

observable_ref = identifier
unit_ref = identifier
frame_ref = identifier

fact_decl =
    simple_fact
  | composite_fact

simple_fact = 'fact' ID 'of' info_ref 'as' value_type
composite_fact = 'fact' ID 'of' info_ref (':' field_decl+)?

info_ref = identifier

enum_decl = 'enum' ID ':' enum_literal+
enum_literal = ID ('=' integer_literal)?

variant_decl =
    'variant' ID ':' (case_member+ otherwise_member? | case_member* otherwise_member)
case_member =
    'case' (enum_literal_ref | integer_literal | bool_literal) (':' field_decl)?
otherwise_member = 'otherwise' ':' field_decl
enum_literal_ref = ID '.' identifier

entity_decl = 'entity' ID (':' field_decl+)?

view_decl = 'view' ID (':' select_clause+)?
select_clause = 'select' entity_member_ref ('as' type)? ':' 'alias' ID)?
entity_member_ref = ID '.' identifier

system_decl = 'system' ID (':' port_decl+)?
port_decl = ('input' | 'output' | 'inout') ID 'of' type

bus_decl = 'bus' ID (':' channel_decl)?
channel_decl = 'channel' ID 'of' type (':' bus_connect_decl+)?
bus_connect_decl = 'connect' port_ref
port_ref = system_ref '.' ID
system_ref = identifier

binding_decl = 'binding' ID 'of' bus_ref 'as' protocol_ref (':' binding_member+)?
binding_member = ('channels' | 'channel' ID) (':' channel_member+)?
channel_member =
    key_decl
  | qos_decl
  | encode_decl
bus_ref = identifier
protocol_ref = identifier
key_decl = 'key' ID (',' ID)*
qos_decl = 'qos' ID? (':' property_assignment+)?
encode_decl = 'encode' identifier ('as' type)?
property_assignment = identifier '=' expression

network_decl = 'network' ID 'of' bus_ref (':' endpoint_decl+)?
endpoint_decl = 'endpoint' ID 'of' binding_ref (':' address_decl)?
address_decl = 'address' (':' property_assignment+)?
binding_ref = identifier

field_decl = ID 'as' type
```

```
type =
    identifier
  | value_type
  | array_type

array_type = '(' type (',' integer_literal)? ')'

value_type =
    'sbyte'
  | 'byte'
  | 'short'
  | 'ushort'
  | 'int'
  | 'uint'
  | 'long'
  | 'ulong'
  | 'single'
  | 'double'
  | 'decimal'
  | 'bool'
  | 'char'
  | 'string'
  | enum_ref

enum_ref = identifier

expression =
    identifier
  | real_literal
  | integer_literal
  | string_literal
  | bool_literal

identifier = ID ('.' identifier)?
ID = id_start (id_part)*
id_start = letter_character | '_'
id_part =
    letter_character
  | decimal_digit
  | connecting_char
  | combining_char
  | formatting_char
decimal_digit = any Unicode character of the class Nd
connecting_char = any Unicode character of the class Pc
combining_char = any Unicode character of the classes Mn or Mc
formatting_char = any Unicode character of the class Cf

integer_literal = decimal_digits
real_literal =
    decimal_digits ('.' decimal_digits)? exponent_part?
  | '.' decimal_digits exponent_part?
decimal_digits = [0-9]+
exponent_part = ('e'|'E')('+'|'-')? decimal_digits

string_literal = '"' character* '"' | '\'' character* '\''
character = char | escape_sequence
escape_sequence = one of \' \" \\ \0 \a \b \f \n \r \t \v

letter_character = any Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
char = any character except new_line
new_line = one of \r \n \r\n

bool_literal = 'true' | 'false'
```

APPENDIX B TEST CASE SIDL DESCRIPTIONS

B.1 Common Data Model

```
namespace Common

info UniqueIdentity
fact UniqueId of UniqueIdentity as ulong

observable Time
observable Position
observable Velocity
observable Acceleration

unit Tick // 1 tick = 100ns
unit Degree
unit Meter
unit MeterPerSecond
unit MeterPerSecondPerSecond

// Coordinated Universal Time
frame UTC
frame EarthCenter

measure TickTime of Time as long:
  units Tick
  frame UTC
  precision 1 // Precision is 1 tick i.e. 100ns

measure Position_Meter_Double of Position as double:
  units Meter
  precision 0.00001
measure Velocity_Meter_Single of Velocity as single:
  units MeterPerSecond
  precision 0.00001
measure Acceleration_Meter_Single of Velocity as single:
  units MeterPerSecondPerSecond
  precision 0.00001
```

B.2 Control Data Model

```
namespace Control

import Common

enum State:
  StandBy
  Start
  Pause
  Run
  Stop

entity Timer:
  Tick as TickTime
  ScenarioTime as TickTime

entity Controller:
  State as State

entity Container:
  Id as UniqueId
  State as State
```

B.3 Entities Data Model

```
namespace Entities

import Common

measure WorldLocation of Position:
  frame EarthCenter
  X as Position_Meter_Double
  Y as Position_Meter_Double
  Z as Position_Meter_Double

measure LinearVelocity of Velocity:
  X as Velocity_Meter_Single
  Y as Velocity_Meter_Single
  Z as Velocity_Meter_Single

measure AccelerationVector of Acceleration:
  X as Velocity_Meter_Single
  Y as Velocity_Meter_Single
  Z as Velocity_Meter_Single

enum DeadReckoning:
  DIS_DRALG_OTHER = 0
  DIS_DRALG_STATIC = 1
  DIS_DRALG_DRM_FPW = 2
  DIS_DRALG_DRM_RPW = 3
  DIS_DRALG_DRM_RVW = 4
  DIS_DRALG_DRM_FVW = 5
  DIS_DRALG_DRM_FPB = 6
  DIS_DRALG_DRM_RPB = 7
  DIS_DRALG_DRM_RVB = 8
  DIS_DRALG_DRM_FVB = 9

entity EntityType:
  EntityKind as byte
  Domain as byte
  Country as ushort
  Category as byte
  Subcategory as byte
  Specific as byte
  Extra as byte

entity EntityDescription:
  Id as UniqueId
  EntityType as EntityType

entity EntityState:
  Id as UniqueId
  LinearVelocity as LinearVelocity
  Location as WorldLocation
  Acceleration as AccelerationVector
  DeadReckoning as DeadReckoning
```

B.4 Ownership Data Model

```
namespace Ownership

import Common

entity Ownership:
  MasterId as UniqueId
  SlaveId as UniqueId
```

B.5 DIS Data Model

```
namespace DIS

import Common

enum PduType:
  DIS_PDUTYPE_OTHER = 0
  DIS_PDUTYPE_ENTITY_STATE = 1

enum EntityKind:
  DIS_EKIND_OTHER = 0
  DIS_EKIND_PLATFORM = 1
  DIS_EKIND_MUNITION = 2
  DIS_EKIND_LIFEFORM = 3
  DIS_EKIND_ENVIRON = 4
  DIS_EKIND_CULTURAL = 5
  DIS_EKIND_SUPPLY = 6
  DIS_EKIND_RADIO = 7
  DIS_EKIND_EXPENDABLE = 8
  DIS_EKIND_SENS_EMIT = 9
  DIS_EKIND_LIMIT = 10

enum PlatformDomain:
  DIS_PLATFORM_DOMAIN_OTHER = 0
  DIS_PLATFORM_DOMAIN_LAND = 1
  DIS_PLATFORM_DOMAIN_AIR = 2
  DIS_PLATFORM_DOMAIN_SURFACE = 3
  DIS_PLATFORM_DOMAIN_SUBSURFACE = 4
  DIS_PLATFORM_DOMAIN_SPACE = 5

enum CharSet:
  DIS_MARKING_CHAR_SET_UNUSED = 0
  DIS_MARKING_CHAR_SET_ASCII = 1

enum DeadReckoning:
  DIS_DRALG_OTHER = 0
  DIS_DRALG_STATIC = 1
  DIS_DRALG_DRM_FPW = 2
  DIS_DRALG_DRM_RPW = 3
  DIS_DRALG_DRM_RVW = 4
  DIS_DRALG_DRM_FVW = 5
  DIS_DRALG_DRM_FPB = 6
  DIS_DRALG_DRM_RPB = 7
  DIS_DRALG_DRM_RVB = 8
  DIS_DRALG_DRM_FVB = 9

entity PduHeader:
  ProtocolVersion as byte
  ExerciseIdent as byte
  PduType as PduType
  ProtocolFamily as byte
  TimeStamp as uint
  Length as ushort
  Padding as (byte, 2)

entity EntityIdent:
  Site as ushort
  AppId as ushort
  EntId as ushort

entity EntityType:
    EntityKind as EntityKind
    Domain as PlatformDomain
    Country as ushort
    Category as byte
    Subcategory as byte
    Specific as byte
```

```
    Extra as byte

entity Vector:
  X as single
  Y as single
  Z as single

entity Position:
  X as double
  Y as double
  Z as double

entity Euler:
  Phi as single
  Theta as single
  Psy as single

entity Marking:
  CharSet as CharSet
  String as (byte, 11)

entity EntityState:
  PduHeader as PduHeader
  EntityIdent as EntityIdent
  ForceIdent as byte
  PartCount as byte
  EntityType as EntityType
  AlternateType as EntityType
  LinearVelocity as Vector
  Location as Position
  Orientation as Euler
  Appearance as uint
  DeadReckoning as DeadReckoning
  Padding as (byte, 15)
  Acceleration as Vector
  AngularVelocity as Vector
  Marking as Marking
  Capabilities as uint
```

B.6 Control Systems

```
import Control

system ControllerSystem:
  input Containers of Container
  output Controller of Controller
  output Timer of Timer

system ContainerSystem:
  input Controller of Controller
  input Timer of Timer
  output Containers of Container
```

B.7 Collision Test Case Systems

```
import Entities

system CollisionSystem:
  inout EntityDescriptions of EntityDescription
  inout EntityStates of EntityState
```

B.8 Ownership Test Case Systems

```
import Entities
```

```
import Ownership

system OwnershipSystem:
  input Ownerships of Ownership
  inout EntityStates of EntityState

system OwnershipService:
  input EntityDescriptions of EntityDescription
  inout Ownerships of Ownership
```

B.9 Gateway Test Case Systems

```
import Entities

system GatewaySystem:
  inout EntityDescriptions of EntityDescription
  inout EntityStates of EntityState
  inout Entities of DIS.EntityState
```

B.10 Control Network

```
import Control

bus ControlBus:
  channel Controller of Controller:
    connect ControllerSystem.Controller
    connect ContainerSystem.Controller

  channel Timer of Timer:
    connect ControllerSystem.Timer
    connect ContainerSystem.Timer

  channel Containers of Container:
    connect ControllerSystem.Containers
    connect ContainerSystem.Containers

binding ControlBusDdsBinding of ControlBus as DDS.Protocol1_2:
  channel Timer:
    qos:
      Reliability.Kind = BestEffort
      Durability.Kind = Volatile
  channel Containers:
    key Id
    qos:
      Reliability.Kind = Reliable
      Durability.Kind = Transient
      History.Kind = KeepLast
  channel Controller:
    qos:
      Reliability.Kind = Reliable
      Durability.Kind = Transient
      History.Kind = KeepLast

network ControlNetwork of ControlBus:
  endpoint ControlBusDds of ControlBusDdsBinding:
    address:
      DomainId = 0
      Partition = "Control"
```

B.11 Collision Test Case Network

```
import Entities

bus EntitiesBus:
```

```
  channel EntityDescriptions of EntityDescription:
    connect CollisionSystem.EntityDescriptions
  channel EntityStates of EntityState:
    connect CollisionSystem.EntityStates

binding entities_dds_binding of EntitiesBus as DDS.Protocol1_2:
  channel EntityDescriptions:
    key Id
    qos:
      Reliability.Kind = Reliable
      Durability.Kind = Transient
      History.Kind = KeepLast
      Ownership.Kind = Exclusive
  channel EntityStates:
    key Id
    qos:
      Reliability.Kind = BestEffort
      Durability.Kind = Volatile
      Ownership.Kind = Exclusive

binding entities_hla_binding of EntitiesBus as HLA.Protocol1516_2010:
  channel EntityDescriptions:
    qos:
      Reliability = Reliable
  channel EntityStates:
    qos:
      Reliability = BestEffort

network entities_demo_network of EntitiesBus:
  endpoint entities_dds of entities_dds_binding:
    address:
      DomainId = 0
      Partition = "Entities"
  endpoint entities_hla of entities_hla_binding:
    address:
      FederationName = "Entities"
```

B.12 Ownership Test Case Network

```
import Ownership

bus ownerships_bus:
  channel Ownerships of Ownership:
    connect OwnershipSystem.Ownerships
    connect OwnershipService.Ownerships

binding ownerships_dds_binding of ownerships_bus as DDS.Protocol1_2:
  channel Ownerships:
    key MasterId, SlaveId
    qos:
      Reliability.Kind = Reliable
      Durability.Kind = Transient
      History.Kind = KeepLast
      Ownership.Kind = Exclusive

network ownerships_demo_network of ownerships_bus:
  endpoint ownerships of ownerships_dds_binding:
    address:
      DomainId = 0
      Partition = "Entities"
```

B.13 Gateway Test Case Network

```
import DIS

bus dis_bus:
```

```
  channel Entities of EntityState:
    connect GatewaySystem.Entities

binding dis_binding of dis_bus as Net.SocketProtocol:
  channels:
    encode PduType as byte
    encode EntityKind as byte
    encode PlatformDomain as byte
    encode CharSet as byte
    encode DeadReckoning as byte

network dis_demo_network of dis_bus:
  endpoint entities of dis_binding:
    address:
      Port = 29000

network entities2_demo_network of EntitiesBus:
  endpoint entities_dds of entities_dds_binding:
    address:
      DomainId = 0
      Partition = "Entities2"
```