A Formal Semantic Framework for Maritime Situation Analysis

by

Amir Yaghoubi Shahir

B.Sc., Azad University, 2012

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Amir Yaghoubi Shahir 2016
SIMON FRASER UNIVERSITY
Spring 2016

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name:	Amir Yaghoubi Shahir
Degree:	Master of Science (Computing Science)
Title:	*A Formal Semantic Framework for Maritime Situation Analysis*
Examining Committee:	**Dr. Ramesh Krishnamurti** (Chair) Professor

Dr. Uwe Glässer
Senior Supervisor
Professor

Dr. Hans Wehn
Supervisor
Adjunct Professor

Dr. William N. Sumner
Internal Examiner
Assistant Professor

Date Defended: 18 April 2016

Abstract

Situation analysis is a process of examining a situation to provide and maintain a state of situation awareness which is critical for dynamic decision-making in responding to real-world situations. Limited surveillance resources constrain maritime domain awareness and compromise safety and security coverage at all times. This calls for innovative intelligent systems for interactive situation analysis to assist marine authorities in their routine surveillance operations. In this research, we use the Abstract State Machine method to formally model and design a precise yet concise situation analysis framework. The backbone of our model is an existing abstract framework. We expand and enrich it by capturing detailed requirements and specifying the precise behavior of its components to process, analyze, and fuse large volumes of marine traffic data received from various sources. To this end, we propose a time series structure for rendering vessel trajectories from real-time data.

Keywords: Maritime Domain Awareness, Time Series Analysis, Situation Analysis and Decision Making, Formal Modeling, Abstract State Machines

This thesis is dedicated to my parents and my grandparents,
for their everlasting love and support.

Acknowledgements

I would first like to thank my senior supervisor, Dr. Uwe Glässer, your support and dedication in guiding me along my path to completing my Masters has been invaluable, you have given me the tools to succeed and for that, there are no words to adequately describe my gratitude. Thank you Dr. Hans Wehn, my supervisor, your advice and continued efforts have helped to shape my research.

Many thanks to Dr. William N. Sumner, and Dr. Ramesh Krishnamurti for their thorough examination of this thesis.

Jeanette, thank you for all your love and support through each step of the way. I would also like to thank to Hamed for guiding me not only through my Masters, but through life. Thanks to my friends at Simon Fraser University, Kamyar, Laleh, Mohammad, Narek, Sara, Ehsan, and Jasneet.

My deepest and warmest thanks to my family, most importantly my parents, for their endless support that has made all of this possible. To all my friends who cannot be mentioned, thank you.

I would also like to acknowledge the administrative and technical staff in the School of Computing Science for their support in the course of this research.

Table of Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgements v

Table of Contents vi

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Motivation 1
1.2 Problem Overview 1
1.3 Thesis Contributions 2
1.4 Thesis Organization 3

2 Marine Traffic Monitoring Systems 5

2.1 Automatic Identification System (AIS) 5
2.1.1 How AIS Works 6
2.1.2 Different Classes and Information 6
2.1.3 Type of Vessel 8
2.1.4 AIS Range 9
2.1.5 AIS Accuracy 9
2.1.6 Drawbacks of AIS 10
2.2 Satellite AIS and How it Works 10
2.3 Marine Radar 11
2.3.1 How Radar Works 11
2.3.2 Drawbacks of Radar 12

3 Abstract State Machines 13

3.1 Lack of Model Designing . . . 13
3.2 The Notion of ASM . . . 14
3.2.1 Basic Concepts . . . 15
3.2.2 Fundamental Concepts . . . 18
3.3 Ground Model . . . 21
3.4 Refinement . . . 22
3.4.1 Formal Definition of Refinement . . . 23
3.5 ASM Systems Engineering Method . . . 24
3.5.1 Distributed ASM . . . 25

4 Time Series Analysis **26**
4.1 Connecting the Dots – From Time Series to Trajectories . . . 26
4.2 Definition of Time Series and its Related Operations . . . 27
4.2.1 Initial Definition: Time Series with One Value per Feature and Precise Timestamps . . . 28
4.2.2 First Refinement: Time Series with a Set of Values per Feature and Precise Timestamps . . . 31
4.2.3 Second Refinement: Time Series with Approximate Timestamps (Practical Approach) . . . 32

5 Maritime Situation Analysis: A Formal Semantic Framework **35**
5.1 Formal Approaches to Situation Analysis (Related Work) . . . 36
5.1.1 JDL (Joint Directors of Laboratories) Data Fusion and State Transition Data Fusion (STDF) Models . . . 36
5.2 Challenges and Key Concepts . . . 39
5.3 Agent Network Structure . . . 40
5.4 Observation Segment . . . 41
5.5 Maritime Situation Analysis Framework . . . 43
5.5.1 Observer Controller . . . 46
5.5.2 Reasoner Controller . . . 48
5.6 Formal Definition of Trajectory . . . 56

6 Implementation **58**
6.1 Implementation of the Framework (CoreASM Part) . . . 59
6.1.1 Network Configuration Module . . . 59
6.1.2 Environment Module . . . 59
6.1.3 Observer and Reasoner Modules . . . 60
6.1.4 Situation Analysis (SA) Module . . . 61
6.2 Implementation of the Framework (Java Part) . . . 62
6.2.1 Data Structure . . . 64

6.2.2 Implementation of Rules 65
6.2.3 Empirical Results 67

7 Conclusions and Future Work **74**
7.1 Future Work 75

Bibliography **76**

Appendix A Implementation Details **80**
A.1 Time Series Operations 80
A.1.1 Horizontal Merge (Initial Definition) 80
A.1.2 Vertical Merge (Initial Definition) 82
A.2 Initialization of Data Structures 84
A.3 Input/Output Manager 85
A.4 Implementation of Rules in Java 87
A.4.1 CleanObservations 87
A.4.2 AffiliateObservations 89
A.4.3 AssociateObservations 91
A.4.4 CombineObservations 93
A.4.5 ResolveInconsistencies 96

List of Tables

Table 2.1 Significant Information Within Class A Reports 7
Table 2.2 Identifying Types of Marine Vessels Using AIS 9

List of Figures

Figure 2.1 The researchers were able to spoof the route of boats [50]. 10

Figure 3.1 Function table consisting of (*location,value*)-pairs [26] 16
Figure 3.2 Classification of ASM functions, relations, locations [8] 19
Figure 3.3 The ASM Refinement Scheme [8] 24

Figure 4.1 Object Tracking 27
Figure 4.2 Example of a Time Series – Head and Body 29
Figure 4.3 Horizontal Merge 29
Figure 4.4 Vertical Merge 30
Figure 4.5 Hybrid Merge 31
Figure 4.6 Time Series with a Set of Values per Feature and Precise Timestamps 31
Figure 4.7 Algorithm for Finding Close Timestamps 34
Figure 4.8 Horizontal Merge with Approximate Timestamps 34

Figure 5.1 The JDL Data Fusion Model [46] 37
Figure 5.2 Information Fusion Model: Adopted from Simplified JDL Model [37] 38
Figure 5.3 Network Structure [37] 41
Figure 5.4 Time Series of an Observation Segment 42
Figure 5.6 Affiliate Observations Rule 47
Figure 5.7 Associate Observations Rule 49

Figure 6.1 Class Diagram of Domain Concepts 63
Figure 6.2 Data Structure of Observation Segments with Two-Layer HashMaps 65

Chapter 1

Introduction

1.1 Motivation

Maritime domain awareness is critical for protecting sea lanes, ports, harbors, fisheries, offshore structures like oil and gas rigs, submarine cable systems, and various other types of critical infrastructure against common threats and illegal activities such as smuggling and piracy. Limited resources constrain maritime domain awareness and compromise full safety and security coverage at all times. This problem is exacerbated in areas with high density and volume of marine traffic, such as waterways in the proximity of major ports, and in vast remote areas lacking close surveillance like the Arctic Ocean. Marine radar, Automatic Identification System (AIS), satellites and other surveillance technologies produce massive volumes of geospatial data. The growing data volume is problematic to analyze in real-time as part of routine surveillance missions to enable proper responses to potential threats and interdict illegal activities for border control and coast guard services [43].

1.2 Problem Overview

Situation analysis (SA) is a process of examining a situation to provide and maintain a state of *situation awareness* [14] which is critical for dynamic decision-making in responding to real-world situations. One of the main aspects of situation analysis is tracking and analyzing the behavior of moving objects (e.g., marine vessels) in vast geographical areas of interest. The movement of each vessel leads to an enormous number of observations, producing maritime surveillance data, which is interpreted as sequences of discrete observations over time that render vessel *trajectories*. Consecutive data points referring to the same vessel form a time series [10] used for analyzing and reasoning about vessel behavior. Generally, these are multi-dimensional time series since each single data point may refer to a number of observed characteristics related to position, heading, speed and several other features. Furthermore, deficiencies in various sources of marine traffic data (e.g., AIS, marine radar, satellites, etc.)

provide another constraint on situation analysis. Rendering a more comprehensive time series of each vessel by fusing data and information from different sources helps to provide a better understanding of unfolding scenarios in the real-world. To this end, time series analysis is required to detect relationships among trajectories of multiple vessels [25].

The systems which conduct real-time accurate situation analysis in the presence of multiple vessels operate in vast geographical areas and need a large number of sensors/observers to obtain accurate situation awareness and to make informed decisions. When the area to be monitored is large, it may often be advantageous to have an adjustable hierarchy of reporting and decision-making components. One of the main contributions of this research is to expand and enrich an existing situation analysis framework proposed in [37] to model and design a robust and extensible SA framework by capturing detailed requirements and specifying the precise behavior of its components in order to process, analyze, and fuse marine traffic data received from various sources. Monitoring areas of interest requires constant communication of relevant information among the framework components. Analyzing real-world situations inevitably needs to deal with complex processing of enormous data. Detecting outliers and cleaning data, building the time series of each vessel by merging data points, resolving inconstancies in the time series, reasoning on vessels' relationships based on trajectories, etc., are parts of this process.

1.3 Thesis Contributions

Maritime surveillance data is interpreted as sequences of discrete observations over time. However, this data is being produced by various types of vessel tracking and monitoring systems. In order to render coherent and consistent vessel trajectories, it is necessary to systematically merge and fuse this data from different sources. The ultimate objective of this thesis is to propose a framework to generate vessel trajectories that can be used for further maritime situation analysis to potentially detect and predict the movement behavior of vessels. To this end;

- We formally propose a structure of *multivariate time series* for marine traffic data along with the definition of two merge operations on this structure. The proposed operations are also refined to satisfy real-world assumptions and requirements.
- We propose a situation analysis framework with precise formal model of its components using Abstract State Machine (ASM) method. The main outcome of the framework is vessels' trajectories.
- We formally define the concept of a "vessel trajectory" and prove some of its characteristics.
- We implement the core of the underlying situation analysis framework in CoreASM and Java to experimentally validate our model and framework.

1.4 Thesis Organization

Below is a brief content overview of each chapter;

Chapter 2: Marine Traffic Monitoring Systems

Critical to the understanding of this work is the foundational understanding of Marine Traffic Monitoring Systems. In this chapter, we examine the systems; how they operate, how they affect, and simultaneously contribute to the formation of a situation analysis framework in the maritime domain. Different marine traffic monitoring systems provide different types of information which represent the different observations found in the real-world. A decent understanding of at least the leading marine traffic monitoring systems, limitations and their capabilities, helps to have a more realistic overview of the situation analysis framework requirements. Because of the importance of the Automatic Identification System amongst other marine traffic monitoring systems, a large portion of this chapter is allocated to discuss different aspect of AIS.

Chapter 3: Abstract State Machines

Abstract State Machine provides a flexible method for mathematical modeling of a system. The ASM ground modeling method is a bridge between intuitive understanding of requirements to a precise yet flexible model. Refining the model improves the design for various purposes such as extension and/or adding more details [8]. In this chapter, we briefly explain the basics of ASM, the concept of *ground model* and *refinement* in ASM, and their formal definitions as background knowledge for later chapters.

Chapter 4: Time Series Analysis

The nature of AIS (and most other marine traffic monitoring systems) is sending information of a given time sequentially. This common feature is a key to connecting all the data points as a time series of all the received information. In order to have a meaningful time series of all the information, we have to find the right type of time series. At the beginning of this chapter, we discuss different types of time series in order to find the correct time series which fits the characteristics of the received information. Secondly, we establish two main different merge operations between time series. Each operation has two refinements, and in the second refinement we introduce an algorithm which enables merging operations to better satisfy real-world requirements. All the initial definitions and refinements are mathematically defined in this chapter.

Chapter 5: Maritime Situation Analysis: A Formal Semantic Framework

In this chapter, we expand and enrich an existing situation analysis framework in [37]. Since there is more than one source of data for marine traffic, we take a look at different information fusion models and methods (such as JDL and STDF) at the beginning of this chapter. The next part of this chapter, we introduce *Observation Segment*—which is an appropriate structure for marine traffic data. The most substantial section of this chapter is the precise design of the framework. As mentioned, the backbone of the proposed situation analysis framework in this research is the existing framework in [37]. All the components in the proposed framework are precisely designed and explained, with some refined to a more detailed level. We use ASM to formally design all the components and refinements.

Chapter 6: Implementation

For experimental evaluation and validation, we implement the core components of the proposed SA framework with the help of CoreASM [15] and Java. CoreASM has a more high-level control, while Java is responsible in dealing with data. This chapter is divided in two sections to explain the CoreASM, and Java sections of implementation. All extra details of the implementations and coding are found in the Appendix A.

Chapter 2

Marine Traffic Monitoring Systems

A vessel traffic service (VTS), which is a marine traffic monitoring system, is utilized in order to periodically transmit reports to remote tracking centres. In order to monitor and provide safety and security of *maritime domain* (MD)[1] against common threats and illegal activities, VTS is established by harbor or port authorities to automatically and accurately monitor marine vessels movements, utilizing a limited number of reports from each vessel. This easily installed and operated system typically uses marine radar and Automatic Identification System (AIS) to track vessel movements and provide safety and efficiency of navigation in a limited geographical area [12].

VTS can also provide information services by broadcasting information at fixed times and intervals or when deemed necessary by the VTS, or at the request of a vessel. Information services can include, for example, reports on the situation of a vessel (e.g., identification and location information), intentions of other traffic (e.g., waterway conditions and weather), or other facts that may influence the vessel's transit.

A modern VTS integrates all of the information compiled from a VTS traffic image from different sources. Integrating all of the information into a single operator working environment is done for ease of use and in order to allow for effective traffic organization and communication [47, 52].

2.1 Automatic Identification System (AIS)

The Automatic Identification System is one of the most important automated tracking and monitoring system used by marine vessels. It has the purpose of improving safety by utilizing the vessel traffic services to gather identification and location information; facilitated by the continuous transmission of data with other nearby vessels, as well as AIS base stations and satellites for collision avoidance of water transport [3, 47]. Heading,

[1] "Maritime domain is all areas and things of, on, under, relating to, adjacent to, or bordering on a sea, ocean, or other navigable waterway, including all maritime-related activities, infrastructure, people, cargo, and vessels and other conveyances" [47].

course over ground (COG), and speed over ground (SOG) would traditionally be provided by AIS equipped marine vessels. Other maneuvering information available is not limited to the following; closest point of approach (CPA), destination, and estimated time of arrival (ETA). AIS normally works in a continuous self-determining mode in its range, regardless of the location of operation (i.e., open seas, coastal or inland areas) [51].

2.1.1 How AIS Works

Every AIS system consists of one very high frequency (VHF) transmitter, two VHF Time division multiple access (TDMA) receivers, one VHF digital selective calling (DSC) receiver, and standard marine electronic communication links to shipboard display and sensor systems. Information provided by AIS equipment can be visualized on monitors and/or an electronic chart display and information system (ECDIS). AIS equipped vessels self-report information consisting of identity, position, and movement to the system in order to calculate reports pertaining to, but not limited to, closest point of approach (CPA) [51]. Moreover, AIS data can potentially be processed automatically in order to create normalized activity patterns for individual vessels. Therefore, navigators of vessels equipped with AIS can avoid collisions with other marine vessel, potential threats (maritime security), and hazards (e.g., sandbars or rocks) [3]. In order to avoid interference problems, and to allow the shifting of channels without diminishing communications from other marine vessels, each station transmits and receives over two radio channels in the Marine VHF band (only the use of one radio channel is necessary) [51]. More advanced AIS equipment uses both frequencies on the Marine VHF channels (Channel A 161.975 Mhz (87B) and Channel B 162.025 Mhz (88B)) [48].

2.1.2 Different Classes and Information

There are varying types of AIS equipment with different technical specifications for the purpose of transmitting and/or receiving data. Usually they include, but are not limited to, vessel-mounted or shore-based equipment. Shore-based AIS equipped with a transceiver (transmit and receive) are able to analysis and control all received reports and/or transmit specific messages to vessels. The most significant of the vessel-mounted equipment are called Class A and Class B. Class A AIS transponders operate using Self-Organizing TDMA (SOTDMA) broadcast mode and transmit information every 2 to 10 seconds while underway (every 3 minutes while at anchor). Static and voyage related vessel information, such as the vessel's name, are transmitted every 6 minutes. They are required to have a receiver, external GPS, heading, and rate of turn indicator. Class A units also transmit and receive safety-related text messages. Class B AIS transponders operate using carrier-sense TDMA (CSTDMA) broadcast mode and transmit information every 30 to 180 seconds. Static data, such as the vessel's name, is transmitted every 6 minutes. A receiver and heading

are optional. Transmitting safety-related text messages is optional and can only be pre-configured into Class B units [41].

To illustrate, Table 2.1 contains the most significant information broadcasted as a position report (168 bits in total) by a Class A AIS unit [51]. The report contains information such as speed over ground, course over ground, rate of turn (ROT), true heading (HDG), latitude, and timestamp, among others [51].

Parameter	Bits	Description
User ID	30	MMSI number which contains at most 9 digits
Navigational Status	4	0 = under way using engine; 1 = at anchor; 2 = not under command; 3 = Restricted maneuverability; 4 = constrained by her draught; 5 = moored; 6 = aground; 7 = engaged in fishing; 8 = under way sail- ing; 9 = reserved for future amendment of navigational status for ships carrying dangerous goods (DG), harm- ful substances (HS), marine pollutants (MP), international maritime organization (IMO) hazard, pollutant category C or high speed craft (HSC); 10 = reserved for future amendment of navigational status for ships carrying DG, HS, MO, IMO hazard, pollutant category A or wing in ground (WIG); 11 = power-driven vessel towing astern (regional use); 12 = power-driven vessel pushing ahead or towing alongside (regional use); 13 = reserved for future; 14 = AIS-SART (active), MOB- AIS, EPORB-AIS; 15 = undefined = default (also used by AIS-SART, MOB-AIS and EPORB-AIS under test)
Rate Of Turn (ROT)	8	The rate of turning right (0 to 126) or left (0 to -126) in degree scale at specific time intervals
Speed Over Ground (SOG)	10	The sum of the water's speed vector and the vessel speed vector in 1/10 knot scale (0 to 102.2 knots)
Position Accuracy	1	The position accuracy (PA) flag should be determined in accordance with 1 = high (≤ 10 m); 0 = low (>10 m) or 0 = default
Longitude	28	Longitude with 1:10000 scale is min (+/-180 degree) such that East is positive and West is negative
Latitude	27	Latitude with 1:10000 scale is min (+/-90 degree) such that North is positive and South is negative
Course Over Ground (COG)	12	Actual direction (0 to 3599) regardless of the course steered and temporary variations in heading around the course
Time Stamp	6	UTC second (0 to 59) when the report was generated by the electronic position system (EPFS)

Table 2.1: Significant Information Within Class A Reports

2.1.3 Type of Vessel

Among all marine vessels there are a limited number which utilize AIS equipment. As a general rule, any self-propelled vessel of 1600 or more gross tons operating in the navigable water of the United States must utilize AIS equipment, except any warship, owned or leased non-commercial vessels by the United States government. Vessels have to utilize specific equipment in order to broadcast either Class A or Class B reports. According to the U.S. Coast Guard Navigation Center, the following list specifies the vessel types and their properties which have either Class A or Class B devices [51];

Class A:

1. A self-propelled vessel of 65 feet or more in length, engaged in commercial service.
2. A towing vessel of 26 feet or more in length and more than 600 horsepower, engaged in commercial service.
3. A vessel that is certificated to carry more than 150 passengers.
4. A self-propelled vessel engaged in dredging operations in or near a commercial channel or shipping fairway in a manner likely to restrict or affect navigation of other vessels.
5. A self-propelled vessel engaged in the movement of
    - Certain dangerous cargo, or
    - Flammable or combustible liquid cargo in bulk

Class A device complying with International Convention for Safety of Life at Sea (SOLAS);

1. A vessel of 300 gross tonnage or more, on an international voyage.
2. A vessel of 150 gross tonnage or more, when carrying more than 12 passengers on an international voyage.

Class B:

1. Fishing industry vessels;
2. Vessels that are certificated to carry less than 150 passengers and that
    - Do not operate in a Vessel Traffic Service (VTS) or Vessel Movement Reporting System (VMRS) area and,
    - Do not operate at speeds in excess of 14 knots[2]; and

[2] Knot is a unit of speed; 1 knot is equal to 1.852 km per hour (or 1.151 mph)

3. Vessels identified engaged in dredging operations.

Each type of vessel has a specific identifier number to be used by ships to report their type. The identifier number consists of different factors such as; type of vessel, type of goods they carry, and number of passengers. As shown in Table 2.2, the first digit of the identifier number indicates the general type of vessel and for instance, for cargo ships, the second digit represents more information about the goods it is carrying [44].

First Digit	General Type
2	Wing In Ground
3	special category
4	High Speed Craft
5	special category
6	Passenger
7	Cargo
8	Tanker
9	Other

Second Digit	Information on Cargo Goods
1	Major Hazard (Haz A)
2	Hazard (Haz B)
3	Minor Hazard (Haz C)
4	Recogniseable Hazard (Haz D)

Table 2.2: Identifying Types of Marine Vessels Using AIS

2.1.4 AIS Range

Shipboard transceivers are connected to an external antenna (on average 15 meters above sea level) which normally sends and/or receives AIS information within a horizontal range of 15-20 nautical miles[3]. Depending on elevation, antenna type, obstacles surrounding antenna and weather conditions, the AIS range can reach to 40-60 nautical miles [34].

2.1.5 AIS Accuracy

The accuracy of AIS information relayed by marine vessels is crucial as many vessels transmit incomplete or inaccurate data. Based on a report of Maritime and Port Authority of Singapore [35] many vessels are transmitting incomplete or incorrect data. As listed below;

1. Incorrect setting of static data, such as the MMSI number and the vessels' dimensions, during the initial installation of the AIS on board.

2. Omission of updating the voyage related data, such as the destination, estimated time of arrival (ETA) and changes in navigational status, by the crew.

3. Incorrect input or processing of dynamic data, such as the position, course, speed and heading, from the vessels sensors.

4. Inconsistent naming of some data, such as the vessels' name and destination.

[3] 1 nautical mile = 1.852 km

2.1.6 Drawbacks of AIS

Although it is complex, yet feasible, to collect data points from AIS equipment in order to study it as a time series. The expected ability to accurately predict the behavior of marine vessels to aid in the improving of maritime security, nonetheless seems beyond the scope of AIS equipment. As reported in [5], AIS equipment is highly vulnerable to hacking (see also Figure 2.1). "Weaknesses in outdated systems could allow attackers to make ships disappear from tracking systems—or even make it look like a large fleet was incoming" [5]. Changing AIS equipment is not the pragmatic solution for improving maritime security with less spoof attempts. We need to use other marine traffic monitoring systems and improve the observing and reasoning process of marine data.

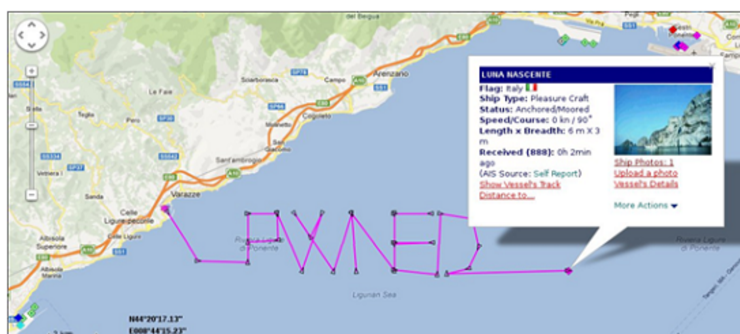


Figure 2.1: The researchers were able to spoof the route of boats [50].

2.2 Satellite AIS and How it Works

Since the 1990's global maritime and port authorities use shore-based AIS to keep track of marine vessels. Shore-based AIS could provide only a limited range of sending and/or receiving data coverage and it could not provide global, open ocean coverage [39]. In recent decades, companies and governments have invested on deploying AIS transceivers on satellites to obtain global coverage. The first step to reach satellite-based AIS monitoring services was attaching two antennas (an AIS VHF antenna, and an Amateur Radio antenna) to the Columbus module (the science laboratory of ISS). Since then, more satellites have been built and launched for the satellite-based AIS-monitoring service purpose (e.g., VesselSat-1 and VesselSat-2 which are among the latest). Canada operates one of the largest AIS satellite networks which provides global coverage using 7 satellites [3].

In order to assess the bigger picture, it is important to note that S-AIS has both its positives and negatives. Case in point, one of the strengths of S-AIS is the ability with which it can be correlated with additional information from other sources such as radar, optical, and more search and rescue (SAR) related tools. Alternatively, a fundamental challenge for AIS satellite operators is the overwhelming number of AIS messages simultaneously

received from a satellite's large reception footprint leaving them inundated. The TDMA radio access scheme defined in the AIS standard creates 4,500 available time-slots (for both channels) per minute. TDMA, such as GSM, is a controlling entity that is used to allocate transmission slots to each user [1, 3].

2.3 Marine Radar

The term RADAR (RAdio Detection And Ranging) designates electronic equipment classed for the tracking and detecting of objects, or targets, at considerable distances. Not only is radar used for detection and ranging of contacts, but it also functions independently of time and weather conditions. This serves to make the radar device one of the most important scientific discoveries and technological advancements to emerge from the 20th Century [38].

The ability of radar to detect a target at great distances, and to locate its position with a high degree of accuracy marks it as ideal for utilization in marine navigation applications. However, radar is rarely ever used alone in a marine setting. For example, in commercial ships, radar is integrated into a much more complex, full system of marine instruments including, chartplotters, sonar, two-way radio communication devices, and emergency locators (SART). Below is a brief summary of sonar and chartplotter, as they along with radar, are crucial marine traffic monitoring equipment.

Sonar
Sonar (SOund Navigation And Ranging) utilizes a technique of sound propagation to navigate, communicate with and/or detects objects on or under the surface of the water.

Chartplotter
Chartplotter is a device that integrates GPS data with an electronic navigational chart (ENC) for the purpose of marine navigation. Along with displaying the ENC, it can also display the position, heading, and speed of a ship and may provide additional information from radar, Automatic Identification System, or other sensors [33].

2.3.1 How Radar Works

Radar stands out from AIS and satellites, as it utilizes its own radio signal to track and detect and is not constrained by self-reporting capabilities dependent on a vessel to transmit. The principle behind radar is simple—relatively short bursts of radio signals, traveling at the speed of light, are transmitted and then reflected off a target and returned as an echo. In more depth, radar functions with the use of a series of *pings* and *echoes*. Thus, radio waves (or pulse) sent out from the radar dish found on top of a vessel, give radar the capability of detecting signals from objects at varied distances (from several meters to kilometres). When an object consequently reflects the signal, the radar computer determines the distance to it and its location. In determining the distance, the signal that is sent out bounces off of

the object it comes into contact with and then registers on the receiver. The receiver then redirects the signal to the computer and measures the time it took for the signal to reflect back. It is crucial that the cycle (a series of equally spaced pulses commonly in durations of about 1 microsecond or less) be completed before the pulse immediately following is transmitted, as the distance to a target is determined by measuring the time required for one pulse to travel to the target and return as a reflected echo [38].

Achieved by way of producing a narrow horizontal beam, marine radar has the fundamental requirement of directional transmission and reception. The performance of marine radar lies within its power and horizontal beam width. However, "radar waves are restricted in the recording of the range of low-lying objects by the radar horizon" [38]. The height of an antenna on a vessel affects the range of the radar horizon, as well as the amount of bending of the radar wave. Moreover, there are only "two groups of radio frequencies allocated by international standards for use by civil marine radar systems" [38]. The first group lies within the X band that corresponds to a wavelength of 2.5-3.75 cm, with a frequency range between 8.0-12 GHz. The second group is found within the S band consisting of wavelength of 7.5-15 cm and a frequency range of 2-4 GHz [33, 38].

2.3.2 Drawbacks of Radar

Weather conditions affect the range of radar causing an inability to detect an object due to inaccurate echoes. The reduction in the effectiveness and strength of waves is known as *attenuation*. Essentially, the absorption and scattering of the wave's energy is the cause of attenuation. The amount of attenuation in radar waves depend on various causes [38]. The radar wavelength (or frequencies) is crucial in the attenuation in a way that radar waves with shorter wavelength (or higher frequencies) have a greater amount of attenuation. The attenuation effect become significance when the wavelengths are shorter than 10cm; therefore, S band suffers less than X band.

Furthermore, different factors (e.g., rain, fog, clouds, hail, snow, dust) in the atmosphere play a key role in the amount of attenuation. "The amount of attenuation caused by these weather factors is dependent upon the amount of water, liquid or frozen, present in a unit volume of air and upon the temperature"[4] [38]. The higher density of particles in the atmosphere increases the amount of attenuation for the radar waves. Particles cause clutter in echoes and subsequently there will be a loss of energy because of the scattering and absorption; therefore, the detection range will decrease [38].

[4] Attenuation takes place to some extent even when radar waves travel through a clear atmosphere [38].

Chapter 3

Abstract State Machines

Abstraction principles and formalization techniques advance modeling of software systems in early design phases. Design is a creative activity calling for abstract models that facilitate reasoning about the key system attributes to ensure that these attributes are well understood and properly established prior to actually building a system. The focus is on specification and validation techniques rather than on formal verification [26].

3.1 Lack of Model Designing

The current software engineering industry needs to be more reliable and efficient. Based on the Standish Group report [2][1], vast numbers of software projects fail because they cannot be delivered on time within the allocated budget. Among those which could meet the time schedule and be completed within the estimated budget, very few satisfy the original specification requirements. Furthermore, testing the code is highly energy consuming and costs more than half of the entire development budget [8]. Yet, in the context of larger, seemingly more complicated projects, there are a number of errors which can cause problems, such as security issues in the software.

The need to have a system engineering method which guides the development of software is widely felt by the industry. A method for filling the gaps from human understanding of the requirements to accurately formulating them, to creating an algorithmic solution and finally, to implementing an executable code. The key for bridging these gaps (especially for rather complicated software) is to develop several levels of abstraction beginning from the ground model in order to maintain a uniform algorithmic view in a single framework. In other words, the defining characteristic of the method utilizes abstraction and refinements.

[1] As stated in the Standish Group report "In the United States, we spend more than $250 billion each year on IT application development of approximately 175,000 projects ... The Standish Group research shows a staggering 31.1% of projects will be cancelled before they ever get completed. Further results indicate 52.7% of projects will cost 189% of their original estimates ... on the success side, the average is only 16.2% for software projects that are completed on- time and on-budget ... projects completed by the largest American companies have only approximately 42% of the originally-proposed features and functions.

Abstract State Machine (ASM) is a high-precision design instrument which is the bridge between the two ends of system development, from requirements capture to detailed design and coding [8]. As stated in [8];

> "The concept of ASMs offers what for short we call *freedom of abstraction*, namely the unconstrained possibility of expressing appropriate abstraction directly, without any encoding detour, to
> – build the ground models satisfying the two parties involved in the system contract, tailoring each model to the needs of the problem as determined by the particular application, which may belong to any of a great variety of conceptually different domains and keeping the models simple, small and flexible (easily adaptable to changing requirements),
> – allow the designer to keep control of the design process by appropriate refinement steps which are fine-tuned to the implementation ideas."

ASM offers the ability to model arbitrarily complex systems. To design such a model, any inessential details are scrapped from the system to obtain a precise and concise model which leaves the core model as a result. By including more details to the core model and refining it, the model of the complex system is gradually built until it reaches an executable version. As a by-product of such a core model and its refinements, designers can analyze and examine the model rationale and structure in order to modify when the requirements are changed or extended [8].[2]

The ASM method has three essential components, namely *the notion of ASM*, *the ground model technique*, and *the refinement principle* which will be explained in the following sections [8].

3.2 The Notion of ASM

Abstract State Machine offer a multipurpose and flexible method for mathematical modeling of a system's behavior and characteristics to bridge the gap between specification methods and computation models. The Abstract State Machine method describes static and dynamic aspects of a systems at any desired level of detail with the necessary degree of exactitude (any desired level of abstraction). ASM builds on a universal computation model by combining two universal concepts; *abstract states* and *transition systems*. Abstract State Machine is known for its versatility in computational and mathematical modeling of all kinds of

[2]On the other hand, core model and its refinements allows users "...to get a general understanding of what the system does, which supports an effective system operator training and is sufficiently exact to prevent as much as possible a faulty system use"; moreover, it helps the maintainers "...to analyse faulty run-time behavior in the abstract model" [8].

sequential, parallel and distributed systems, with practical applications [7, 21, 23, 24, 26, 40, 45].[3]

The general idea of Abstract State Machine is defined in two rounds. The first round, Section 3.2.1 illustrates the basic computation model by means of a simple sample program for sorting linear data structures consisting of a single transition rule. This is to provide an applied perspective at an intuitive level of understanding. Additional rule constructs for forming more complex programs will be described informally. In the second round, Section 3.2.2 explains fundamental concepts of *synchronous* and *asynchronous* computation models in some detail, building on common concepts and notions from discrete mathematics and computational logic. Specifically, first-order predicate calculus and first-order language will provide us with a universal syntactic and semantic foundation for expressing the descriptive elements of Abstract State Machines [26].

3.2.1 Basic Concepts

In computing science, every algorithm uses a fixed set of instructions to access and manipulate the content of the data structures (reading and writing values in memory). At any given time in the course of executing an algorithm, both data structures and control structures define the *state* of the algorithm. The set of states of an algorithm can be finite or infinite, and normally contains a non-empty subset of distinguished *initial states* which hold special conditions [26].

Each computation step of an algorithm potentially changes the value of one or more locations of one or more data structures, thus altering the successive state. Accordingly, stepwise execution of an algorithm, also called *run*, is defined as a sequence of consecutive state transitions such that each step performed on a given state S_i, for $i \in \{0, 1, \ldots\}$, results in a next state S_{i+1}, where the difference between S_i and S_{i+1} is referred to as Δ_{S_i},

$$S_0 \xrightarrow{\Delta_{S_0}} S_1 \xrightarrow{\Delta_{S_1}} S_2 \xrightarrow{\Delta_{S_2}} \cdots$$

As mentioned, a run normally starts in a distinguished initial state S_0, but depending on the nature of an algorithm, a run may or may not terminate and reach a distinguished final state (an algorithm can be finite or infinite) [26].

ASM Representation of Algorithms

As stated in [26], describing an algorithm in terms of an Abstract State Machine means to;

1. Model states of the algorithm by a collection of data structures abstractly defined in terms of *sets*, *functions* (operations) and *relations*

[3] See also the overview in [8].

2. Specify the state transition behavior by means of a *program* defining how these data structures can be accessed and manipulated.

The choice of data structures which form the machine states, determine the level of abstraction of the algorithm. The machine program, together with the set of distinguished initial states, implicitly describes the possible machine runs in terms of a state transition system that inductively defines the set of states that are reachable from a distinguished initial state. In general, both the set of initial states and the set of reachable states may be infinite [26].

Basic Rule Types

ASM rules must be able to describe any sequential, parallel or distributed algorithm regardless of the abstraction level, as explained below;

The canonical ASM rule consists of a basic update instruction of the form

$$f(t_1, t_2, \ldots, t_n) := t_0,$$

where f denotes an n-ary *dynamic function* and each t_i, for $0 \leq i \leq n$, refers to a term that can be evaluated in a given state. Intuitively, one may perceive a dynamic function $f : X^n \rightarrow Y$ as being represented by a function table where each row of the table associates a location of f consisting of a sequence of argument values $a_1, a_2, \ldots, a_n$ with a function value b as illustrated in Figure 3.1.

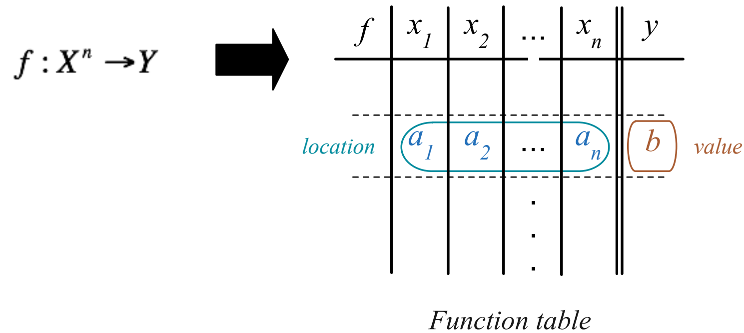


Figure 3.1: Function table consisting of (*location,value*)-pairs [26]

A basic update instruction as shown above specifies a *pointwise* update of the interpretation of a dynamic function f as represented by the underlying function table in any given state. That is, the value (say b) associated with the location referred to by $t_1, t_2, \ldots, t_n$ (say $a_1, a_2, \ldots, a_n$), is to be overwritten with the value of t_0 (say b', with $b' \neq b$). While all the terms t_i are evaluated over the current state, the result of the update operation takes effect only in the next state [26].

As stated in [26], for the description of complex instructions the composition of basic update instructions into complex ASM rules with preconditions is inductively defined by means of the rule constructors explained below;

Rule Constructors

Together with the basic update instruction, the *rule constructors* listed below form a comprehensive language for expressing ASM programs. While additional rule types may be introduced as syntactic abbreviations they do not add to the expressiveness of the language.

$$R \equiv \textbf{if} \quad e \quad \textbf{then} \quad R_1 \quad \textbf{else} \quad R_2$$

$$\begin{aligned} R \equiv\ & \textbf{do in parallel} \\ & \quad R_1 \\ & \quad R_2 \end{aligned}$$

$$\begin{aligned} R \equiv\ & \textbf{forall} \quad x \in S : \phi(x) \\ & \quad R_1(x) \end{aligned}$$

$$\begin{aligned} R \equiv\ & \textbf{choose} \quad x \in S : \phi(x) \\ & \quad R_1(x) \end{aligned}$$

$$\begin{aligned} R \equiv\ & \textbf{extend}\ S\ \textbf{with} \quad x \\ & \quad R_1(x) \end{aligned}$$

Abstract semantics [26];

- A rule R of the form "**if** e **then** R_1 **else** R_2" specifies a condition e as a boolean-valued expression, together with two subrules denoted by R_1, R_2. The evaluation of e over a given state always yields a defined value, depending on whether R_1 or R_2 is selected.
- A rule R of the form "**do in parallel** R_1 R_2" states the composition of the rules R_1, R_2 such that the two rules are to be applied in parallel. Since this situation is common, the key word "**do in parallel**" is often dropped.[4]
- A rule R of the form "**forall** $x \in S : \phi(x)$ $R_1(x)$" states that for all the elements x in a given *finite* set S for which a specified condition $\phi(x)$ holds a copy of the rule $R_1(x)$ in which x is instantiated to such an element is to be applied in parallel.

[4]Parallelism naturally occurs in algorithms whenever two or more instructions or operations that are enabled in a given state do not interfere with each other. However, this property is often ignored by programming languages that unnaturally impose linearization only because of their underlying execution model.

- A rule R of the form "**choose** $x \in S : \phi(x)\ R_1(x)$" randomly chooses some element x in a given *finite* set S for which a specified condition $\phi(x)$ holds and, if any such element exists, applies the rule $R_1(x)$ in which x is instantiated to the chosen element; otherwise it does nothing.

- A rule R of the form "**extend** S **with** $x\ R_1(x)$" introduces a new element as the value of the variable x into the specified set S and applies the rule $R_1(x)$ in which x is instantiated to the chosen element. This rule is needed to dynamically introduce new resources at runtime.

3.2.2 Fundamental Concepts

There are two prominently used ASM computation models which are explained in greater detail in this section.

The first model, *basic ASM*, befits computations with synchronous parallel actions performed by a single computational agent and captures the notion of parallel algorithm in a comprehensive sense as discussed in [6, 26]. The second model, *Distributed ASM*, allows any number of autonomously operating computational agents to cooperatively perform asynchronously distributed computations [26].

Synchronous Parallelism

As stated in [26], a basic ASM M is defined as a tuple of the form $(\Sigma,\ \mathcal{I},\ \mathcal{R},\ P_M)$ where;

- Σ is a *vocabulary* consisting of finitely many function symbols, each with a fixed arity (number of arguments);

- $\mathcal{I}$ is a set consisting of one or more initial states defined over Σ;

- $\mathcal{R}$ is a collection of transition rules, and

- $P_M \in \mathcal{R}$ is a distinguished rule, called the *main rule* or the Program of machine M.

The following definitions (Signature − Transition) were established formally in [8]. As such, we have not taken the liberty to make any changes from the original.

Signature

A *signature* Σ is a finite collection of function names. Each function name *f* has an *arity*, a non-negative integer. Nullary function names are called *constants*. Function names can be *static* or *dynamic*. The dynamic functions are further classified according to Figure 3.2. Every ASM signature is assumed without further mention to contain the static constants *undef*, *true*, *false*.

A signature is also called a *vocabulary*. The arity of a function name is the number of arguments that function takes. Be aware that the interpretation of dynamic nullary functions can change from one state to the next, so that they correspond to the variables of programming.

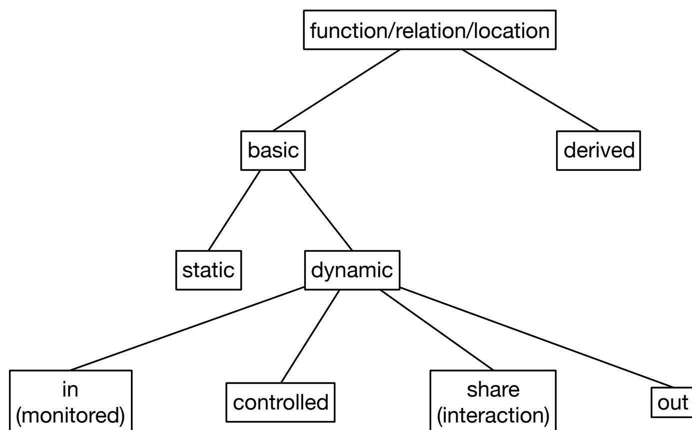


Figure 3.2: Classification of ASM functions, relations, locations [8]

State

A state $\mathfrak{A}$ for Σ is a many-sorted structure that yields a valid interpretation of Σ. That is, for each function symbol $f : X^n \mapsto Y$ in Σ, with $n = 0, 1, 2, \ldots$, the interpretation $f^{\mathfrak{A}} : X^n \mapsto Y$ is consistent with the function symbol and all related constraints that apply. Function symbols can be *static* or *dynamic*. Interpretations of dynamic function symbols can change from state to state, whereas static function symbols do have a fixed interpretation regardless of the state of M. In addition to common static symbols like *true*, *false* and the usual Boolean, arithmetic and algebraic operations, which are all static, Σ contains a distinguished static symbol *undef* representing the 'undefined' value.[5]

Location

A *location* of $\mathfrak{A}$ is a pair $(f,\ (a_1, \ldots, a_n))$, where *f* is an *n*-ary function name and $a_1, \ldots, a_n$ are elements of $|\mathfrak{A}|$. The value $f^{\mathfrak{A}}(a_1, \ldots, a_n)$ is called the *content* of the location in $\mathfrak{A}$. The *elements* of the location are the elements of the set $\{a_1, \ldots, a_n\}$.

A state $\mathfrak{A}$ can be viewed as a function that maps the locations of $\mathfrak{A}$ to its contents.

[5]Partial functions are modeled on top of total functions by setting undefined values to the distinguished value *undef*.

Update and Update Set

Evaluation of a transition rule in a given state yields a finite set of *updates* of the form $\langle (f, \langle a_1, \ldots, a_n \rangle), v \rangle$, where f is an n-ary function symbol in Σ, $\langle a_1, \ldots, a_n \rangle$ is a location in $Dom(f)$ and v is a value in $Ran(f)$. An update set is consistent if no location occurs more than once.

Transition

A state transition from a given state

$$S_i \xrightarrow{\Delta_{S_i}} S_{i+1}$$

such that S_{i+1} is obtained from S_i, for $i \geq 0$, by firing Δ_{S_i} on S_i, where Δ_{S_i} denotes a finite set of updates computed by evaluating agent programs over S_i. Firing an update set means that all the updates in the set are fired simultaneously in one atomic step. The result of firing an update set is defined if and only if the set does not contain any conflicting (inconsistent) updates.

Asynchronous Multi Agent ASM

An arbitrary finite number of agents each of which is executing a basic ASM independently in its own local state, is called *asynchronous* multi-agent ASM.[6] The main considerable issue with asynchronous ASM is agents may run with different clock, data, and duration of execution. Since there is no order to the moves of functions, it is unreasonable to define a global state where moves are executed to specific changes; therefore, the notion of *local* move for each agent in each state is defined [26].

Real Time

In a given state S of M, the global time (as measured by an external clock) is given by a monitored nullary function *now* taking values in a linearly ordered domain $\mathsf{TIME} \subseteq \mathbb{R}$. Values of *now* increase monotonically over runs of M. Additionally, ∞ represents a distinguished value of TIME, such that $t < \infty$ for all $t \in \mathsf{TIME} - \{\infty\}$. Finite time intervals are given as elements of a linearly ordered domain DURATION.

The ASM concept of *physical time* is defined orthogonally to the concept of state transition, flexibly supporting a wide range of time models, also including continuous time [26, 27]. A frequently used model is that of distributed real-time ASM with time values ranging over positive real numbers [26].

[6] A run of an asynchronous multi agent ASM is also called a *partially ordered run* of ASM.

3.3 Ground Model

Derivation and deduction of requirements is a well-known struggle and error prone part of the system development process. Building ground model ASM helps to have a better solution on three major difficulties of requirements capture. As stated in [8];

> "Requirements capture is largely a *formalization task*, namely to realize the transition from natural language problem descriptions – which are often incomplete and interspersed with misleading details, partly ambiguous or even inconsistent – to a sufficiently precise, unambiguous, consistent, complete and minimal description, which can serve as a basis for contract between the customer or domain expert and software designer. We use the term *ground model* for such an accurate description resulting from the requirements elicitation . . . "

According to [8], the formalization task helps the three major difficulties found in building the ground model by way of model inspection, verification, and testing, as briefly explained below;

1. The first formalization problem is one of *language and communication*. This entails a precise and understandable language for the application domain expert (the contractor) and the system designer. It must be created in such a way as to allow the ground model to accurately express the relevant features of the given application domain, all the while doing so naturally. As such, the modeling language needs to provide a general data model combined with a function model, as well as an accompanying interface to the environments.

2. The second problem which formalization helps solve is a *verification method*. This is do to the lack of mathematical means which prove the accuracy of the passage from an informal to the precise description. To get past this issue, the designer must be able to formally check the completeness and internal consistency of the model, along with the consistency of different system views. One possible solution to this problem is ASM, which allows customization of the ground model to resemble the structure of real-world problems, and in so doing, makes it possible to inspect its correctness and analyze its completeness with respect to the problem to be solved.

3. The third problem is *validation*. Simulating the ground model for running relevant scenarios of the system is often considered a system acceptance test plan (prior to coding). The ground model ASM can be used for static testing, where the code is inspected and compared to the specification, and also it can be used for dynamic testing, where the execution results are compared.

So far the over all concept of ground model has been explained but the general mathematical definition of the notion of ground model has not been characterized. Based on [8],

the mathematical definition of the notion of ground model is characterized by the following underlying properties. The ground model has to be:

- *precise* at the appropriate level of detailing to assure the accuracy of requirements yet *flexible* to be modifiable for any further use such as extending or adapting to different application domains.

- *simple and concise* to be easily understandable by both parties involved, domain expert (the contractor) and the system designer. Furthermore, the simplicity in ground model avoids any unnecessary and unrelated encoding through its abstract structure of real-world problem.

- *abstract (minimal) yet complete.* Minimality means that the ground model is abstracted in relevant details. Any details which are related to future design modification and/or extension should not influence. Completeness means a ground model should contain and present every semantically relevant features and parameters (e.g., benefits and obligations which are mentioned in contract).

- *validatable* which means it should be capable of simulating the relevant scenarios of the system. As a consequence, the executable ground model helps to understand and analyze the behavior of the system and validate it.

- equipped with a *precise semantical foundation* as a prerequisite for analysis and a reliable tool for development.

3.4 Refinement

The second building block of the ASM method for designing models is refinement. Refining a model to improve the system design (for various purposes such as extension and/or adding more details or modifying for the use of other application domains) is a long established and characterized idea in designing and developing a system. One of the tenets of refinement notions in the literature is:

> "*principle of substitutivity*: it is acceptable to replace one program by another, *provided* it is impossible for user of the program to observe that the substitution has taken place" [13].

Due to the primary focus of literature on the principle of substitutivity, there are resulting restrictions which limit the range of applicability. Based on [8] these restrictions mainly are as follows:

- Restriction to *certain forms of programs* which basically means the sequence of operations should occur in the refined model as it occurs in

the abstract counterpart; therefore, the refined model will be structurally equivalent to the abstract counterpart.

- Restriction to programs with *monolithic state operations* impedes any possibility of modifying elements in any state. This restriction reduce the combination of local effects on the refined model and basically prevents having more difficult formal specification of programs than the initial description of the program.
- Restriction to *observations interpreted as pairs of input/output sequences or of pre-post-states* ensures that the input/output representation at the abstract model and the refined model are equivalent.
- Restriction to *logic or proof-rule-oriented* refinement schemes. Tailoring refinement schemes to fit a priori fixed proof principles quickly leads to severe restrictions of the design space.

Alternatively, refinements in the ASM method give a more open framework due to the freedom of abstraction. The availability of the arbitrary abstract structure in ASM, which mirrors the underlying notion of state, provides a way to find a well-defined mapping from the abstract structure to a refined and more concrete one. The mapping should be consistent with a more detailed state and more involved computation in a way that the intended equivalency between this corresponding is explicitly defined and proved (Figure 3.3).

(0) show that an implementation S^* satisfies a desired property P^* the ASM method allows the designer to

(1) build an abstract model S,

(2) prove a possibly abstract from P of the property in question to hold under appropriate assumptions for S,

(3) show S to be correctly refined by S^* and assumptions to hold in S^*.

3.4.1 Formal Definition of Refinement

The formal definition of correctness and completeness of refinement as stated in [8] is found below;

Definition – Correct Refinement

Fix any notions $\equiv$ of states and of initial and final states. An ASM M^* is a correct refinement of an ASM M if and only if for each of M^*-run $S_0^*,\ S_1^*, \ldots$ there is an M-run $S_0,\ S_1, \ldots$ and sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S_{j_k}^*$

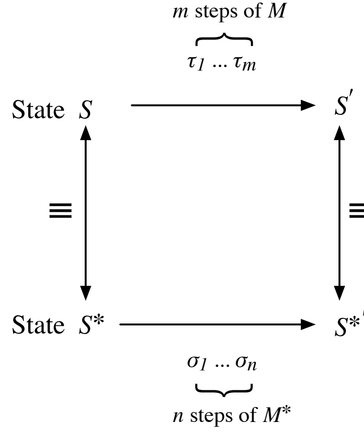


Figure 3.3: The ASM Refinement Scheme [8]

for each k and either
– both runs terminate and their final states are the last pair of equivalent states, or
– both runs and both sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ are infinite [8].

Definition – Complete Refinement

M^* is a complete refinement of M if and only if M is a correct refinement of M^* [8].

3.5 ASM Systems Engineering Method

The ASM method [8] aims at industrial system design and development by integrating precise high-level, problem-domain oriented modeling into the design and development cycle, and by systematically linking abstract models down to executable code [26].

The method consists of three essential elements: *i)* capturing the requirements into a precise yet abstract operational model, called a *ground model* ASM, *ii)* systematic and incremental refinement of the ground model down to the implementation, and *iii)* experimental model validation through simulation or testing at each level of abstraction [26]. This process emphasizes freedom of abstraction as a guiding principle, meaning that original ideas behind the design of a system can be expressed in a direct and intuitive way so as to enable system designers to stress on the essential aspects of design rather than encoding insignificant details [26].

Starting from a ground model and applying the process of step-wise refinement [8], a hierarchy of intermediate models can be created that are systematically linked down to

the implementation. At each step, the refined model can be validated and verified to be a correct implementation of the abstract model. The resulting hierarchy serves as a design documentation and allows one to trace requirements down to the implementation [26].

3.5.1 Distributed ASM

A distributed ASM (DASM) M_D is defined by a dynamic set AGENT of autonomously operating computational *agents*, each executing a basic ASM independetly. This set may change dynamically over runs of M_D, as required to model a varying number of computational resources. Agents of M_D interact with one another by reading and writing shared locations of a global machine state, and also M_D typically interacts with its operational environment which is the part of the external world visible to M_D, and formally represented by controlled and monitored functions [26].

Of particular interest are *monitored functions*, read-only functions controlled by the environment. A typical example is the abstract representation of global system time in terms of a monitored function *now* taking values in a linearly ordered domain TIME. Values of *now* increase monotonically over runs of M_D [8, 26].

Chapter 4

Time Series Analysis

The International Maritime Organization (IMO) defines *maritime domain awareness* (MDA) as "the effective understanding of anything associated with the maritime domain that could impact the security, safety, economy, or environment" [28]. To avoid any such threat to the maritime domain, an accurate understanding of the current state of the maritime domain is required. Comprehensive fusion of data and information from pertinent sources (e.g., AIS, S-AIS, radar, satellite images, humans, and beyond) helps to improve the awareness of the current state and trend of the maritime domain. Data will be received periodically to monitor and analyze maritime activities in order to identify any anomalies and/or threats. Due to the multiplicity of the sources, data from monitoring real-world situations is often subject to inconsistency in addition to corrupted values, missing values or noise [37]. With these deficiencies it would be even more complicated to analyze and extract any meaningful behavior. Therefore, all received data at any point in time needs to be integrated in a coherent and consistent way to have a better understanding of the current state (and build a history of states). Representing all the received data in an ordered sequence as a time series, and working with time series is the approach of this work for analyze data in the maritime domain.

4.1 Connecting the Dots – From Time Series to Trajectories

A time series is a collection of a sequential series of data points (i.e., observations) measured at successive points over time. The study of these data points, in order to extract meaningful feature, behavior, characteristics, and statistics, is called time series analysis. Processing time series to reach a state of situation awareness is one of the goals of this thesis. To reach this goal, we need to have a reasonable understanding of received observations from maritime domain (Chapter 2), and also time series, in order to describe how a marine vessel moves through space as a function of time (called *trajectory*) and analyze trajectories.

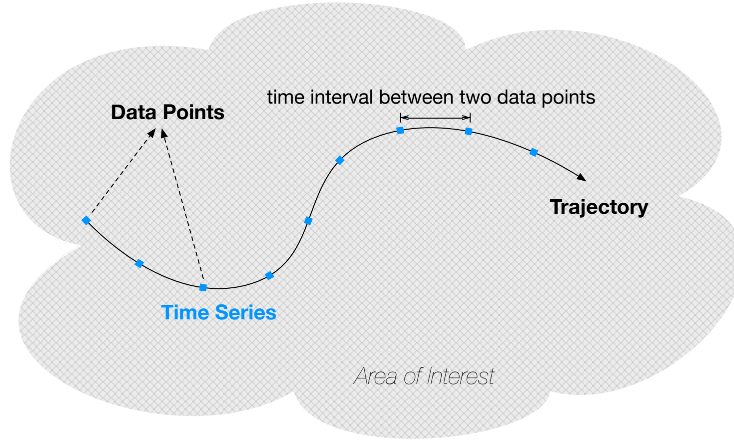


Figure 4.1: Object Tracking

4.2 Definition of Time Series and its Related Operations

A simple time series is an ordered sequence of data points (e.g., values) of a single variable at equally spaced time intervals [13]. The study of these data points, in order to extract meaningful feature, behavior, and statistics, is called time series analysis [49]. But all time series are not simple time series. Depending on the filed of study, time series have a different variety of characteristics, as described below;

- A time series which has the values of more than one variable at each point in time (i.e., *timestamp*) is called a *multivariate time series.*
- A time series may not be *uniform*, which means it does not have equal space time intervals between any two successive data points.
- There is temporal ordering between data points in any simple time series. However, if there is more than one source for receiving data, the temporal order between observations can be defined as *full order* or *partial order.*
- A time series can be defined as *continuous* or *discrete* based on the nature of the observations (i.e., taken continuously through time or at specific times).
- A time series can also be defined as *stochastic* or *deterministic.* Most time series are stochastic as the future is partly determined by the past and the current state [10].

Due to the fact that observations are received from different sources of marine traffic monitoring systems, we need to select the corresponding time series (among all the possible variety of time series) in order to match observations' characteristics and behaviors received from the maritime domain. Each observation contains different types of information and

they are received randomly at any given timestamp.

Corollary – Sensors are monitoring the real-world environment, therefore they are dealing with discrete and stochastic multivariate time series which may not be uniform.

Assumption – A global clock for all sensors (e.g., AIS receivers) is assumed in this research which causes a full temporal order between all timestamps (in the same or different time series).

As stated in [10], the objectives of time series analysis can be categorized in four distinct categories; *description*, *explanation*, *prediction* and *control*. Description is a means of plotting the data points to describe the trend of changes over time which may occur gradually or dramatically. Moreover, the graph (time plot) can highlight the occurrence of outliers. Explanation, or modelling, is to explain observations which have two or more variables in one time series. In this research, observations on maritime vessels have more than one variable (e.g., location, course, speed). Prediction signifies the function of predicting the future behavior of time series depending on given data points. Time series can be predictable if the successive observations are dependent. Control is usually the strategy, or plan, to have control of the changes of a time series. Based on the time plot, and studying the behavior and characteristic of the time series, a controller then takes the optimal control strategy [10].

The rest of the section gradually builds upon the definition of the concept of time series that will be used for the proposed framework.

4.2.1 Initial Definition: Time Series with One Value per Feature and Precise Timestamps

As illustrated in Figure 4.2, Time series $\mathcal{T}$ of a single *object* (e.g., marine vessel) is defined with two sets;

- $Head(\mathcal{T})$ is a tuple of *features* with a unique order, $(f_{1_\mathcal{T}}, f_{2_\mathcal{T}}, \ldots, f_{n_\mathcal{T}})$, one of which must be the global *timestamp*.

$$Head(\mathcal{T}) = \{timestamp\} \ \cup \text{Features}$$

- $Body(\mathcal{T})$ is a sorted set of tuples, $\{\tau_i\}$, with the same order of $Head(\mathcal{T})$, for which, each has one value (or null), $(v_{f_{1_{\tau_i}}}, v_{f_{2_{\tau_i}}}, \ldots, v_{f_{n_{\tau_i}}})$, of every feature. The $Body(\mathcal{T})$ is sorted by *timestamp*s.

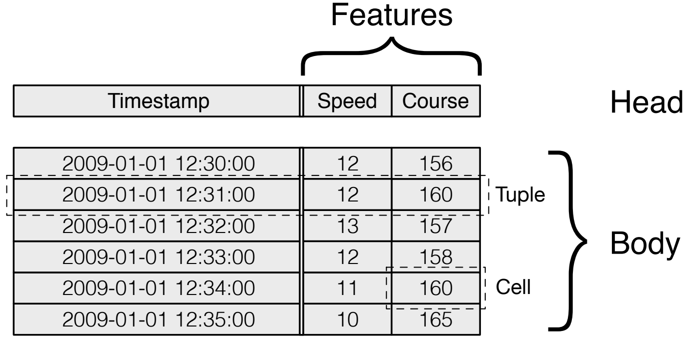


Figure 4.2: Example of a Time Series – Head and Body

Horizontal Merge

Assume there are two time series, $\mathcal{T}_i$ with n features and $\mathcal{T}_j$ with m features, the horizontal merge $\bowtie(\mathcal{T}_i, \mathcal{T}_j)$ is defined as;

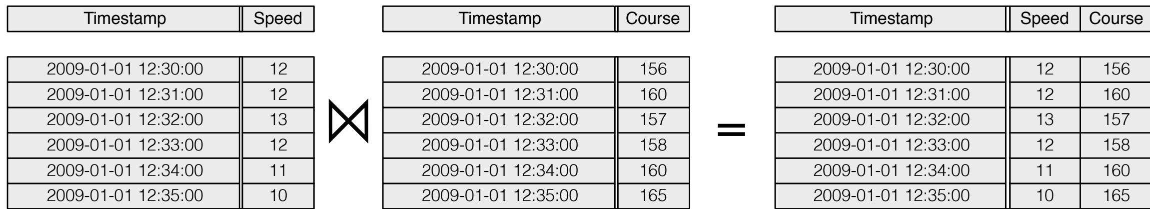
Preconditions:

$$Head(\mathcal{T}_i) \cap Head(\mathcal{T}_j) = \{timestamp\}$$
$$\forall \tau_i \in Body(\mathcal{T}_i), \exists \tau_j \in Body(\mathcal{T}_j) : timestamp_{\tau_i} = timestamp_{\tau_j} \ \wedge$$
$$\forall \tau_j \in Body(\mathcal{T}_j), \exists \tau_i \in Body(\mathcal{T}_i) : timestamp_{\tau_j} = timestamp_{\tau_i}$$

$$Head(\bowtie(\mathcal{T}_i, \mathcal{T}_j)) \stackrel{\text{def}}{=} (f_{1_{\mathcal{T}_i}}, f_{2_{\mathcal{T}_i}}, \ldots, f_{n_{\mathcal{T}_i}}, f_{1_{\mathcal{T}_j}}, f_{2_{\mathcal{T}_j}}, \ldots, f_{m_{\mathcal{T}_j}})$$

$$Body(\bowtie(\mathcal{T}_i, \mathcal{T}_j)) \stackrel{\text{def}}{=} \{(v_{f_{1_{\tau_i}}}, v_{f_{2_{\tau_i}}}, \ldots, v_{f_{n_{\tau_i}}}, v_{f_{1_{\tau_j}}}, v_{f_{2_{\tau_j}}}, \ldots, v_{f_{m_{\tau_j}}}) \mid \tau_i \in Body(\mathcal{T}_i),$$
$$\tau_j \in Body(\mathcal{T}_j) \text{ which } timestamp_{\tau_i} = timestamp_{\tau_j}\}$$

Note that, horizontal merge can be applied on more than two time series (e.g., $\bowtie(\mathcal{T}_i, \mathcal{T}_j, \ldots, \mathcal{T}_k)$).



Timestamp	Speed
2009-01-01 12:30:00	12
2009-01-01 12:31:00	12
2009-01-01 12:32:00	13
2009-01-01 12:33:00	12
2009-01-01 12:34:00	11
2009-01-01 12:35:00	10

⋈

Timestamp	Course
2009-01-01 12:30:00	156
2009-01-01 12:31:00	160
2009-01-01 12:32:00	157
2009-01-01 12:33:00	158
2009-01-01 12:34:00	160
2009-01-01 12:35:00	165

=

Timestamp	Speed	Course
2009-01-01 12:30:00	12	156
2009-01-01 12:31:00	12	160
2009-01-01 12:32:00	13	157
2009-01-01 12:33:00	12	158
2009-01-01 12:34:00	11	160
2009-01-01 12:35:00	10	165

Figure 4.3: Horizontal Merge

Vertical Merge

Assume there are two time series, $\mathcal{T}_i$ and $\mathcal{T}_j$, the vertical merge $⧖(\mathcal{T}_i, \mathcal{T}_j)$ is defined as;

Preconditions:

$Head(\mathcal{T}_i) = Head(\mathcal{T}_j)$

$\forall \tau_i \in Body(\mathcal{T}_i), \nexists \tau_j \in Body(\mathcal{T}_j) : timestamp_{\tau_i} = timestamp_{\tau_j}$

$$Head(⧖(\mathcal{T}_i, \mathcal{T}_j)) \stackrel{\text{def}}{=} Head(\mathcal{T}_i)$$

$$Body(⧖(\mathcal{T}_i, \mathcal{T}_j)) \stackrel{\text{def}}{=} Body(\mathcal{T}_i) \cup Body(\mathcal{T}_j)$$

Note that, vertical merge can be applied on more than two time series (e.g., $⧖(\mathcal{T}_i, \mathcal{T}_j, \ldots, \mathcal{T}_k)$).

Timestamp	Speed	Course
2009-01-01 12:30:00	12	156
2009-01-01 12:31:00	12	160
2009-01-01 12:32:00	13	157
2009-01-01 12:33:00	12	158

⧖

Timestamp	Speed	Course
2009-01-01 12:34:00	11	160
2009-01-01 12:35:00	10	165

=

Timestamp	Speed	Course
2009-01-01 12:30:00	12	156
2009-01-01 12:31:00	12	160
2009-01-01 12:32:00	13	157
2009-01-01 12:33:00	12	158
2009-01-01 12:34:00	11	160
2009-01-01 12:35:00	10	165

Figure 4.4: Vertical Merge

Hybrid Merge

The application of hybrid merge, $\boxtimes(\mathcal{T}_i, \mathcal{T}_j)$, occurs in the instance of two time series, $\mathcal{T}_i$ and $\mathcal{T}_j$, not satisfying at least one precondition of both horizontal merge and vertical merge. Essentially, hybrid merge is applying both horizontal and vertical merges on two time series and dealing with null cells (i.e., a null cell is the result of not satisfying preconditions in horizontal/vertical merge). In other words, merging two time series horizontally, without the identical sets of timestamps, requires adding the tuples with missing timestamps and null values for all the features, or merging two time series vertically, without identical heads, requires adding the missing features and set null values for all timestamps of the features. Note that, hybrid merge can be applied for more than two time series (e.g., $\boxtimes(\mathcal{T}_i, \mathcal{T}_j, \ldots, \mathcal{T}_k)$).

Timestamp	Speed
2009-01-01 12:30:00	12
2009-01-01 12:33:00	12
2009-01-01 12:34:00	11

$\boxtimes$

Timestamp	Course
2009-01-01 12:30:00	156
2009-01-01 12:31:00	160
2009-01-01 12:32:00	157
2009-01-01 12:33:00	158

=

Timestamp	Speed	Course
2009-01-01 12:30:00	12	156
2009-01-01 12:31:00	NULL	160
2009-01-01 12:32:00	13	NULL
2009-01-01 12:33:00	12	158
2009-01-01 12:34:00	11	160
2009-01-01 12:35:00	10	NULL

Figure 4.5: Hybrid Merge

4.2.2 First Refinement: Time Series with a Set of Values per Feature and Precise Timestamps

In this refinement, each feature can have a set of values in the body of a time series (see Figure 4.6).

Timestamp	Speed	Course
2009-01-01 12:30:00	12	{156, 158}
2009-01-01 12:31:00	{12, 13}	157
2009-01-01 12:32:00	{12, 13}	{158, 159}
2009-01-01 12:33:00	12	158
2009-01-01 12:34:00	{11, 12}	{160, 161}
2009-01-01 12:35:00	{10, 11}	164

Figure 4.6: Time Series with a Set of Values per Feature and Precise Timestamps

Horizontal Merge

If two time series, $\mathcal{T}_i$ with m features and $\mathcal{T}_j$ with n features, have some similar features in their heads, the horizontal merge $\bowtie(\mathcal{T}_i, \mathcal{T}_j)$ (with *similar timestamps* precondition: $\forall \tau_i \in Body(\mathcal{T}_i), \exists \tau_j \in Body(\mathcal{T}_j) : timestamp_{\tau_i} = timestamp_{\tau_j} \wedge \forall \tau_j \in Body(\mathcal{T}_j), \exists \tau_i \in Body(\mathcal{T}_i) : timestamp_{\tau_j} = timestamp_{\tau_j}$) is defined as;

$$
\begin{aligned}
&Head(\bowtie(\mathcal{T}_i, \mathcal{T}_j)) \stackrel{\text{def}}{=} (f_{1_{\mathcal{T}_i}}, f_{2_{\mathcal{T}_i}}, \ldots, f_{m'_{\mathcal{T}_i}}, f_{1_{\mathcal{T}_j}}, f_{2_{\mathcal{T}_j}}, \ldots, f_{n'_{\mathcal{T}_j}}) \text{ which} \\
&\quad \forall f_{x_{\mathcal{T}_i}(1 \le x \le m')} \in Head(\mathcal{T}_i) \implies \exists! f_{y_{\bowtie(\mathcal{T}_i, \mathcal{T}_j)}} \in Head(\bowtie(\mathcal{T}_i, \mathcal{T}_j)) \wedge \\
&\quad \forall f_{x_{\mathcal{T}_j}(1 \le x \le n')} \in Head(\mathcal{T}_j) \implies \exists! f_{y_{\bowtie(\mathcal{T}_i, \mathcal{T}_j)}} \in Head(\bowtie(\mathcal{T}_i, \mathcal{T}_j))
\end{aligned}
$$

$$
\begin{aligned}
&Body(\bowtie(\mathcal{T}_i, \mathcal{T}_j)) \stackrel{\text{def}}{=} \{(v_{f_{1_{\tau_i}}}, v_{f_{2_{\tau_i}}}, \ldots, v_{f_{m'_{\tau_i}}}, v_{f_{1_{\tau_j}}}, v_{f_{2_{\tau_j}}}, \ldots, v_{f_{n'_{\tau_j}}}) \text{ which} \\
&\quad \tau_i \in Body(\mathcal{T}_i) \wedge \tau_j \in Body(\mathcal{T}_j) \wedge timestamp_{\tau_i} = timestamp_{\tau_j} \\
&\quad \text{and } \exists y, f_{y_{\mathcal{T}_i}} = f_{z_{\mathcal{T}_j}} \wedge f_{y_{\mathcal{T}_i}} = f_{x_{\bowtie(\mathcal{T}_i, \mathcal{T}_j)}} \\
&\quad \implies v_{f_{x_{\bowtie(\mathcal{T}_i, \mathcal{T}_j)}}} = \{v_{f_{y_{\mathcal{T}_i}}}, v_{f_{z_{\mathcal{T}_j}}}\},\ v_{f_{x_{\bowtie(\mathcal{T}_i, \mathcal{T}_j)}}} \in Body(\bowtie(\mathcal{T}_i, \mathcal{T}_j))
\end{aligned}
$$

In other words, if there is a similar feature in the heads of two time series, the value of that feature in the body of the merged time series, is the union of related values in each time series.

Vertical Merge

If two time series, $\mathcal{T}_i$ and $\mathcal{T}_j$ with k features, have a number of similar timestamps in their bodies, the vertical merge $\rotatebox{90}{$\bowtie$}(\mathcal{T}_i, \mathcal{T}_j)$ (with precondition: $Head(\mathcal{T}_i) = Head(\mathcal{T}_j)$) is defined as;

$$Head(\rotatebox{90}{$\bowtie$}(\mathcal{T}_i, \mathcal{T}_j)) \stackrel{\text{def}}{=} Head(\mathcal{T}_i)$$

$$\begin{aligned}
&Body(\rotatebox{90}{$\bowtie$}(\mathcal{T}_i, \mathcal{T}_j)) \stackrel{\text{def}}{=} Body(\mathcal{T}_i) \cup Body(\mathcal{T}_j) \text{ s.t.} \\
&\quad \forall \tau_m, \tau_n : \tau_m \in Body(\mathcal{T}_i) \land \tau_n \in Body(\mathcal{T}_j) \land timestamp_{\tau_m} = timestamp_{\tau_n} \\
&\quad \implies \tau_{\rotatebox{90}{$\bowtie$}(m,n)} = (\{v_{f_{1_{\tau_m}}}, v_{f_{1_{\tau_n}}}\}, \{v_{f_{2_{\tau_m}}}, v_{f_{2_{\tau_n}}}\}, \ldots, \{v_{f_{k_{\tau_m}}}, v_{f_{k_{\tau_n}}}\}) \text{ which} \\
&\quad \tau_{\rotatebox{90}{$\bowtie$}(m,n)} \in Body(\rotatebox{90}{$\bowtie$}(\mathcal{T}_i, \mathcal{T}_j)) \text{ and } timestamp_{\tau_{\rotatebox{90}{$\bowtie$}(m,n)}} = timestamp_{\tau_m}
\end{aligned}$$

In other words, if there are similar timestamps in the bodies of two time series, the value of each feature with that timestamp in the body of the merged time series, is the union of the values in each time series.

4.2.3 Second Refinement: Time Series with Approximate Timestamps (Practical Approach)

Due to real-world constraints, it is not realistic to assume two time series, which are produced from two different sources, have two identical sets of timestamps. As a consequence, the merged time series will have a relatively big number of null cells. This practical approach helps to have a merged time series with less null cells. If $timestamp_{\tau_m}$ and $timestamp_{\tau_n}$ are *close* enough, they can be considered as occurring at the same timestamp (they can be considered as *similar timestamps*).

Definition – Close Timestamps

$timestamp_{\tau_m}$ and $timestamp_{\tau_n}$ (which $\tau_m \in Body(\mathcal{T}_i)$ and $\tau_n \in Body(\mathcal{T}_j)$) are close if and only if:

$$|timestamp_{\tau_n} - timestamp_{\tau_m}| < \delta \quad \text{(which } \delta \text{ is constant)}$$

Therefore, in the practical approach, horizontal merge does not require the precondition in the first refinement. Likewise, in the vertical merge, having close timestamps (rather than similar timestamps) is enough for the union of the values of the same feature in the bodies of two time series. Another challenge for merging two time series in a real-world situation is that the data stream will receive from sources endlessly. Therefore, the algorithm which

finds the close pair of timestamps should be an online algorithm. The purpose of this algorithm is to reduce the number null cells by finding close timestamps; however, it does not minimize the number of null cells and finds the optimal close timestamps as data is received.

Algorithm: Based on the assumption in Section 4.2, all timestamps are full temporal ordered; therefore, timestamps of two (or more) time series, which should be merged, can be sorted with simple comparisons as the data is received. In a sorted list, based on the definition of close timestamps, finding close timestamps needs another comparison between each two successive timestamps. In this algorithm, each timestamp has a flag of "True" or "False" which shows if it is already marked as a close timestamp of another timestamp or not. Initially, the flag of every timestamp is set to False. Also, if n time series are merging, each timestamp will be close to at most *n-1* different timestamps (which means each of them is from a different time series).

Preconditions: To find the close timestamps of a timestamp like $timestamp_{\tau_m}$, at least one piece of data with a timestamp greater than $timestamp_{\tau_m} + \delta$ should have already been received; in other words, time interval $(timestamp_{\tau_m},\ timestamp_{\tau_m} + \delta)$ should be available.

Algorithm Steps:

1. A timestamp, $timestamp_{\tau_m}$, with the False flag selects a limited number of both preceding and successive timestamps through the list, such that either it selects the *n-1* nearest, and different, timestamps whose flags are set as False (before itself and/or after itself) or it exceeds the threshold δ (in time intervals of $(timestamp_{\tau_m} - \delta,\ timestamp_{\tau_m})$ and/or $(timestamp_{\tau_m},\ timestamp_{\tau_m} + \delta)$).

2. If before (and/or after) $timestamp_{\tau_m}$ there is more than one timestamp belonging to one time series, the selected timestamp will be the nearest to $timestamp_{\tau_m}$.

3. In each side of the $timestamp_{\tau_m}$ (before and after it) there will be at most *n-1* selected different timestamps. If in both sides there is a selected timestamp of one time series, the nearest one will be marked as close timestamp. Otherwise, in both sides only one selected timestamp of a time series exists which will be marked as close timestamp to $timestamp_{\tau_m}$.

4. If $timestamp_{\tau_m}$ finds any close timestamps, the flags of all timestamps which are marked as close timestamps to $timestamp_{\tau_m}$, and $timestamp_{\tau_m}$ will be turned to True. Otherwise, the flags of $timestamp_{\tau_m}$ stay as False.

5. The following timestamp after $timestamp_{\tau_m}$ with the False flag, will repeat the steps as explained above.

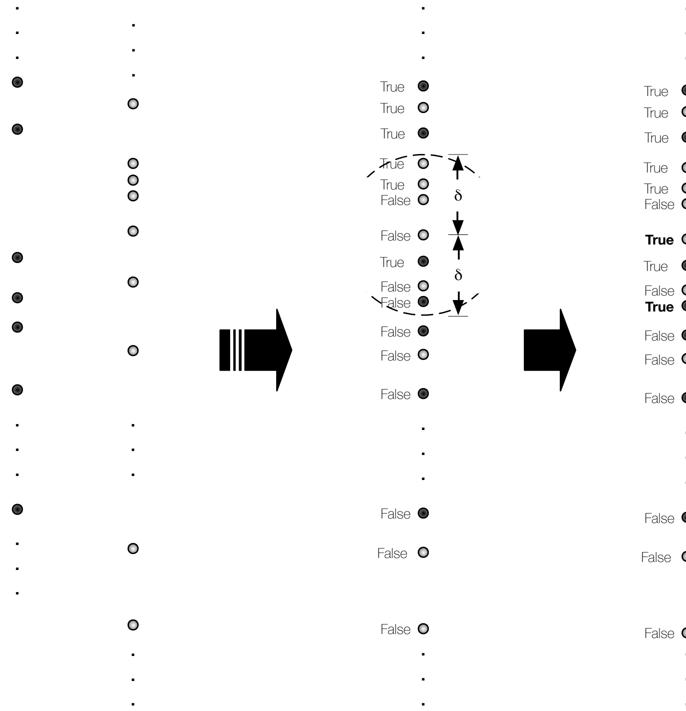


Figure 4.7: Algorithm for Finding Close Timestamps

Horizontal/Vertical Merge

Merging two (or more) time series in this refinement (both horizontally and vertically) is the same as the first refinement with the difference being that instead of *similar timestamps* in the first refinement, *close timestamps* will be considered. Note that, close timestamps are different from each other; therefore, the average of all of them is the timestamp of related tuple in the body of the merged time series.

Timestamp	Speed	Course
2009-01-01 12:30:13	12	156
2009-01-01 12:31:23	12	160
2009-01-01 12:31:58	13	157
2009-01-01 12:33:19	12	158
2009-01-01 12:34:30	11	160
2009-01-01 12:35:04	10	165

$\bowtie$

Timestamp	Speed	Course
2009-01-01 12:29:59	12	158
2009-01-01 12:31:19	13	161
2009-01-01 12:33:05	13	159
2009-01-01 12:33:46	12	158
2009-01-01 12:34:32	12	161
2009-01-01 12:34:50	11	164

=

Timestamp	Speed	Course
2009-01-01 12:30:06	12	{156, 158}
2009-01-01 12:31:21	{12, 13}	{160, 161}
2009-01-01 12:31:58	13	157
2009-01-01 12:33:12	{12, 13}	{158, 159}
2009-01-01 12:33:46	12	158
2009-01-01 12:34:31	{11, 12}	{160, 161}
2009-01-01 12:34:57	{10, 11}	{165, 164}

Figure 4.8: Horizontal Merge with Approximate Timestamps

Merging two time series by definition (in Initial Definition and First Refinement) is not subjected to any errors. However, the Second Refinement may be subject to error depending on the threshold (δ). The algorithm may not find an acceptable group of close timestamps if δ is not well chosen. Since δ is an arbitrary constant, the error of the algorithm is dependent on its value. If δ approaches *0*, the *close timestamps* approaches *similar timestamps*.

Chapter 5

Maritime Situation Analysis: A Formal Semantic Framework

Situation analysis (SA) refers to a process of examining a situation and its related factors to have a perception of environment [14]. Situation analysis provides and maintains a state of *situation awareness* [37], which is defined as below based on [14];

> "the perception of the elements in the environment within a volume of time and space, the comprehension of their meaning and projection of their status in near future"

Due to the fact that multiple surveillance resources (e.g., satellites and radars) are monitoring vast geographical areas in the maritime domain, an intelligent and interactive situation analysis model helps to provide situation assessment and situation awareness for dynamic decision-making. Thus, a reliable computational SA framework is vital as the foundation of complex decision-making processes. As a result, the development of an intelligent system tailored to assist and guide human operators by supporting their decision-making tasks allows for the focus to lie in critical situations outside the scope of the routine activities [37].

The abstract framework in [37] is the backbone of the proposed maritime situation analysis framework in this chapter. By capturing more details in response to real-world situations, we propose a situation analysis framework (particularly focused on the maritime domain) which uses situational evidences from varied surveillance resources.

Cleaning, processing, and transforming raw time series of marine data into vessel trajectories is challenging, especially if there is more than one data source. Thus, information fusion models and methods (e.g., JDL or STDF) play a pivotal role here.

5.1 Formal Approaches to Situation Analysis (Related Work)

We briefly discuss related works on existing situation analysis models. As stated in [37], there are three major approaches to formalizing situation analysis concepts and models, which are characterized as follows:

1. Methods based on general purpose formal logics such as [4];
2. Methods based on machine learning and ontologies such as [11];
3. Distinct situation analysis models and methods such as Dynamic Case-Based Reasoning [29] and State Transition Data Fusion (STDF) [32].

In the following section, we focus on the STDF model, which is the basis of our proposed situation analysis framework. In fact, STDF is an extension of the JDL (Joint Directors of Laboratories) Data Fusion model [9, 46].

5.1.1 JDL (Joint Directors of Laboratories) Data Fusion and State Transition Data Fusion (STDF) Models

As stated in [46], the initial definition of data fusion proposed by data fusion lexicon in 1987 is as follows;

> "A process dealing with the association, correlation, and combination of data and information from single and multiple sources to achieve refined position and identity estimates, and complete and timely assessments of situations and threats, and their significance. The process is characterized by continuous refinements of its estimates and assessments, and the evaluation of the need for additional sources, or modification of the process itself, to achieve improved results" [53].

However, a more concise definition is proposed in [46];

> "Data fusion is the process of combining data to refine state estimates and predictions."

Since 1987, various revisions of data and information fusion models have been proposed; however, the main objective of all these models is to provide distinction among different fusion levels. In this section, the JDL model proposed in [46] and its main five levels are elaborated (see also Figure 5.1). Below, the main responsibilities of each level is reported from [46];

- Level 0 – Sub-object Data Assessment: Estimation and prediction of signal/object observable states on the basis of pixel-/signal-level data association and characterization. Level 0 assignment involves hypothesizing the presence of a signal (i.e., of a common source of sensed energy) and estimating its state.

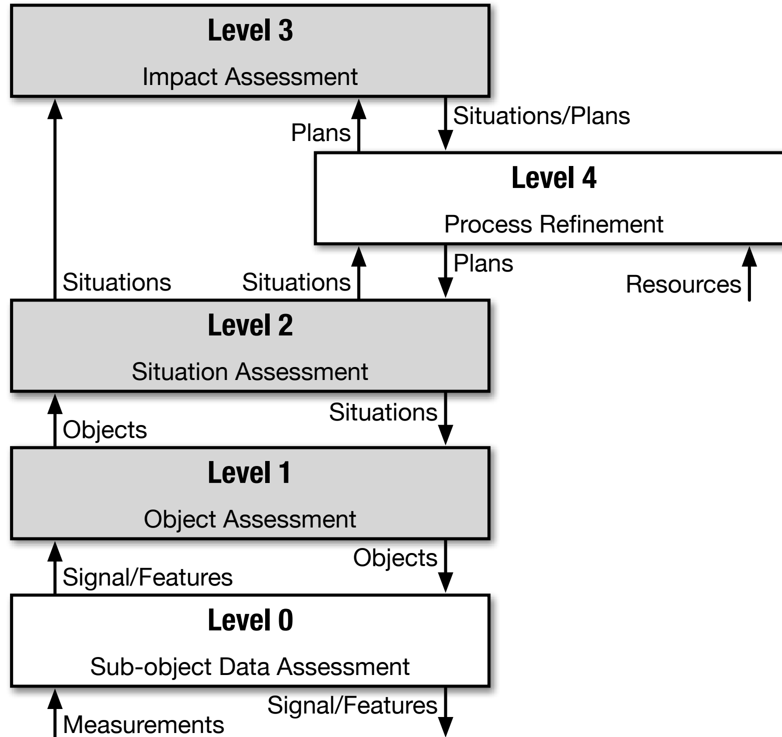


Figure 5.1: The JDL Data Fusion Model [46]

- Level 1 – Object Assessment: Estimation and prediction of entity states on the basis of observation-to-track association, continuous state estimation (e.g., kinematics) and discrete state estimation (e.g., target type and ID). Level 1 assignments involve associating reports (or tracks from prior fusion nodes) into association hypotheses, for which we use the convenient shorthand "tracks". Each such track represents the hypothesis that the given set of reports is the total set of reports available to the system referencing some individual entity.

- Level 2 – Situation Assessment: Estimation and prediction of relations among entities to include force structure and cross-force relations, communications and perceptual influences, physical context, and so forth. Level 2 assignment involves associating tracks (i.e., hypothesized entities) into aggregations. The state of the aggregate is represented as a network of relations among its elements. Any variety of relations is considered—physical, informational, perceptual, organizational—given that it is appropriate to the given system's mission. As the class of relationships estimated and the numbers of interrelated entities broaden, Steinberg, Bowman, and White tend to use the term situation for an aggregate object of estimation.

- Level 3 – Impact Assessment: Estimation and prediction of effects on situations of planned, estimated, or predicted actions by the participants, to include interactions

between action plans of multiple players (e.g., assessing susceptibilities and vulnerabilities to estimated or predicted threat actions, given one's own planned actions). Level 3 assignment is usually implemented as a prediction function, drawing particular kinds of inferences from Level 2 associations. Level 3 fusion estimates the "impact" of an assessed situation (i.e., the outcome of various plans as they interact with one another and with the environment). The impact estimate can include likelihood and cost or utility measures associated with potential outcomes of a player's planned actions.

- Level 4 – Process Refinement: Adaptive data acquisition and processing to support mission objectives. Level 4 processing involves planning and control, not estimation. Level 4 assignment involves assigning tasks to resources.

The STDF model provides a unification of both sensor and higher-level fusion across three main levels [32]; Object Assessment, Situation Assessment, and Impact Assessment (highlighted in Figure 5.1). Both JDL and STDF are functional models. The STDF model is less abstract than the JDL model, and seeks to demonstrate a unifying framework across all three levels. Figure 5.2 illustrates our interpretation of the common ground in terms of an abstract fusion model adopted from the JDL [9], and its extension State Transition Data Fusion (STDF) model [32].

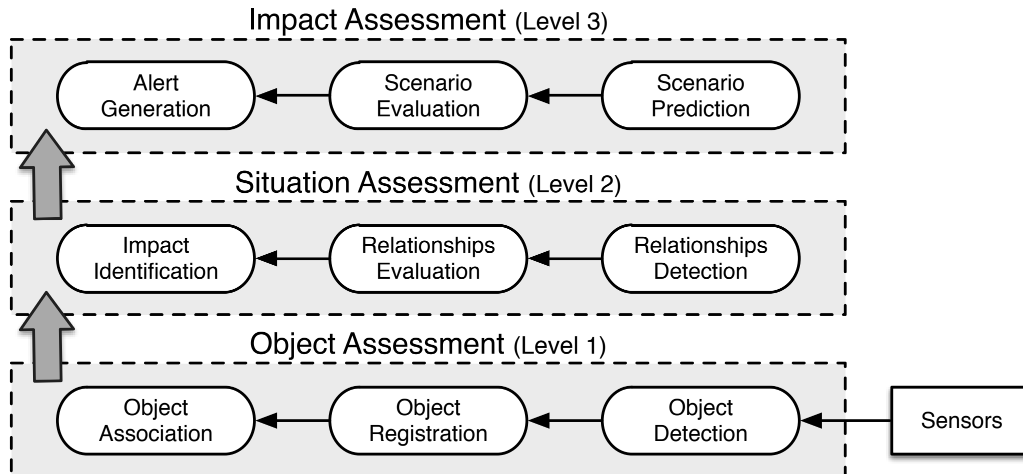


Figure 5.2: Information Fusion Model: Adopted from Simplified JDL Model [37]

It is composed of the Object Assessment phase, where the system detects objects in the area of interest, identifies their properties and their behavior. The system then passes the control on to the Situation Assessment phase, where it detects the relationships between objects, reasons about the properties of such relationships, and detects abnormal behavior. Finally, acquiring awareness of the current situation, in the Impact Assessment phase the system predicts possible scenarios that may unfold over time, evaluates most likely scenarios and possible impacts of those scenarios. The Object Assessment phase returns sets of

statements, each corresponding to one object in the area of interest; the Situation Assessment phase returns sets of sets of statements, each set corresponding to a situation, which typically involves more than one object; and the Impact Assessment phase returns a set of scenarios and a corresponding evaluation for each scenario. The latter is a sequences of situations itself.

As studied in [4, 37] proposes an operational decision support model by employing reasoners that are associated with specific logics. However, a situation analysis system needs to reason about data over time and this aspect is not supported by the applied reasoners. Other approaches and methods specifically focusing on early stages of problem formulation and design are explored in [16, 20, 30]. There is a necessity for integrated approaches to design situation analysis and decision support systems for complex domains [31]. As stated in [31, 36];

> "Large, complex systems are hard to evolve without undermining their dependability. Often change is disproportionately costly, ... system architectures are pivotal in meeting the above challenge. ... First, dependability properties tend to be emergent, and are much more readily modeled and controlled at an architectural level".

Existing (formal) frameworks designed and proposed for situation analysis models mainly focus on specific theoretical aspects. As mentioned in [37], practical needs call for adaptive and evolutionary analysis and design methods that encompass iterative modeling to experimentally validate the essential aspects of design through rapid prototyping of executable models in early phases.

5.2 Challenges and Key Concepts

Best engineering practice calls for a system to be modeled prior to construction.[1]

When modeling a situation analysis system, it is crucial to consider the impact factors of the real-world or physical environment in which the system operates. Various sensors are monitoring areas of interest to collect situational evidence of events. This results in a series of discrete observations, and consequently time series of collected observations. Due to weather conditions, vulnerability of equipment, and technological limitations (as discussed in Chapter 2), observations often are inaccurate, incomplete or even invalid. Such inconsistency, corruption, and uncertainty in data leads to serious challenges. With this in mind, one of the challenges of our proposed situation analysis framework is fusing data and information from multiple diverse sources to develop and render a coherent consistent global picture of a complex real-world situation as it unfolds over time [37].

[1] Uwe Glässer and Hamed Yaghoubi Shahir, *ASM/*CoreASM *Tutorial* ([26])

The foundation of our proposed situation analysis framework is adopted from [32], while this framework mainly focuses on object assessment and situation assessment layers (see also Figure 5.2). We define the key concepts and vocabulary of our proposed situation analysis framework accurately as follow;

- **Objects** are all entities that can be observed in the real-world (e.g., maritime domain).
- **Observation** refers to a single data point in a time series, typically associated with a single object in the real-world. It is assumed that each observation is associated with a timestamp.
- **Observation Segment** refers to a structure used to preserve the information (i.e., a collection of observations) about a single object and its related time series over a certain period of time.[2] Details of the observation segment structure are explained in Section 5.4.
- **Trajectory** refers to a comprehensive time series that describes how an object moves through space as a function of time. It is important to note that additional object characteristics beyond geospatial and kinematic aspects can be part of the time series. The most important geospatial and kinematic features are location, course over ground, speed over ground, and rate of turn.

5.3 Agent Network Structure

The structure of agents defines the relations and the interactions between different conceptual components (*sensors*, *observers*, and *reasoners*.) in the generic architecture of the framework. As illustrated in Figure 5.3, a set of sensors is connected to an observer (sensors of each observer provide observations as the input to the observer) and a set of observers is connected to a reasoner.

The framework has flexibility in its structuring of the level of hierarchy, as a result a fourth layer (not illustrated in Figure 5.3) can potentially be added which consists of the human decision-maker who depends on the information from reasoners.

The definitions of sensors, observers, and reasoners are adopted from [37] as follows:

- **Sensors** are interfaces of the system with the environment. They are located in the external world in such a way as to cover the area of interest. Often, multiple sensors are used to provide a better understanding of the real-world situation as a combination of different observations at any given state. Each sensor belongs to only one observer and its observations are the input of its corresponding observer. Typical sensor types

[2] Aggregation of one or more time series as part of one observation segment where the goal is to merge these time series into one coherent and consistent resulting time series.

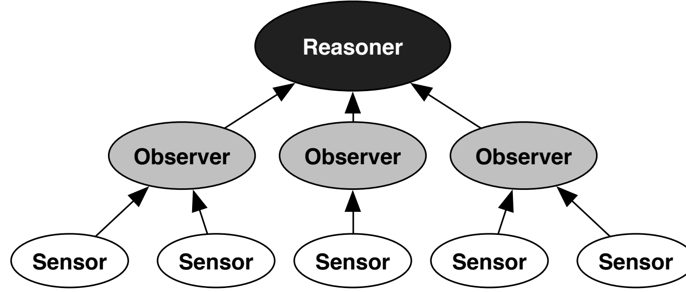


Figure 5.3: Network Structure [37]

include AIS transceivers, radar, satellites, sensor units on aircrafts, the human eye, etc. Sensors are potentially prone to errors, thus, each sensor has a *confidence value* which indicates the reliability of its observation.

- **Observers** are computational agents which process its corresponding sensors' observations. Each observer generates local observations by combining its own sensors' output. Each observer belongs to only one reasoner. Observers may be mobile or stationary.

- **Reasoners** are computational agents which have a global view of the environment by processing the output of its belonging observers. In addition, reasoners typically have access to additional contextual and background information.

$$
\begin{aligned}
&\mathit{reasoner} : \textsc{Observer} \mapsto \textsc{Reasoner} \\
&\mathit{observers} : \textsc{Reasoner} \mapsto \textsc{Set}(\textsc{Observer}) \\
&\mathit{observer} : \textsc{Sensor} \mapsto \textsc{Observer} \\
&\mathit{sensors} : \textsc{Observer} \mapsto \textsc{Set}(\textsc{Sensor})
\end{aligned}
$$

5.4 Observation Segment

In this section, we define the structure of an *observation segment* which is used to preserve the information about each observable entity (i.e., object). Domains and universes along with, corresponding functions, represent observation segments and are defined in the following;

Universes and Domains

AGENT ≡ SENSOR ∪ OBSERVER ∪ REASONER
OBJECT // An object is a vessel which is identified by an *ID* (e.g., MMSI).
TIMESERIES
FEATURE
TUPLE
RECORD
VALUE
AGENTID
OBSERVATIONSEGMENT // or OS as abbreviation

Each observation segment has four elements: owner, object, time series, and status.

Functions Related to the Definition of Observation Segment

owner : OBSERVATIONSEGMENT ↦ AGENT
object : OBSERVATIONSEGMENT ↦ OBJECT
timeSeries : OBSERVATIONSEGMENT ↦ TIMESERIES
status : OBSERVATIONSEGMENT ↦ {UNDEF, CLEANED, MERGED, UNIFIED, FUSED, CONSISTENT}

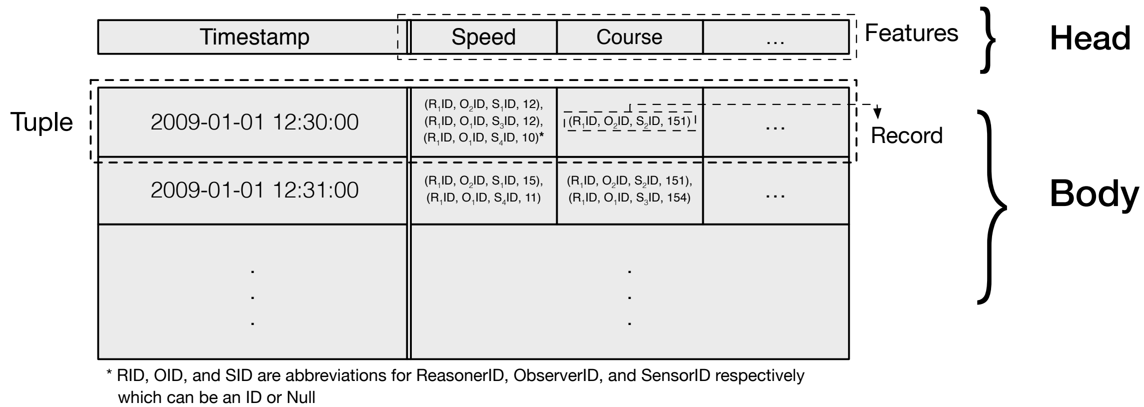


Figure 5.4: Time Series of an Observation Segment

- **Owner** is an agent which belongs to either one of the Sensors or Observers or Reasoners with an agent ID.
- **Object** is a vessel which is identified by an *ID* (e.g., MMSI).
- **Time Series** in an observation segment can possibly contain multiple values (or null) in a cell of its body due to the multiplicity of sensors, observers, and reasoners. The time series in an observation segment is a multivariate time series with the difference being that instead of one or more values (or null) for every cell in the $Body(\mathcal{T})$, there

are one or more 4-tuples such as *(ReasonerID , ObserverID, SensorID, Value)* called *record* as in Figures 5.4.

- **Status** is a multifaceted flag to show where each observation segment is located through out the system as it is illustrated in Figure 5.5.

Functions related to the time series of each observation segments are defined in the following;

Functions Related to the Definition of Time Series of an Observation Segment

$$
\begin{aligned}
&\textit{head} : \textsc{TimeSeries} \mapsto \textsc{List}(\textsc{Feature})\\
&\textit{body} : \textsc{TimeSeries} \mapsto \textsc{List}(\textsc{Tuple})\\
&\textit{tupleTimestamp} : \textsc{Tuple} \mapsto \textsc{Time}\\
&\textit{record} : \textsc{TimeSeries} \times \textsc{Time} \times \textsc{Feature} \mapsto \textsc{Set}(\textsc{Record})\\
&\textit{startTimestamp} : \textsc{TimeSeries} \mapsto \textsc{Time}\\
&\textit{endTimestamp} : \textsc{TimeSeries} \mapsto \textsc{Time}
\end{aligned}
$$

$$
\begin{aligned}
&\textit{value} : \textsc{Record} \mapsto \textsc{Value}\\
&\textit{sensorID} : \textsc{Record} \mapsto \textsc{AgentID}\\
&\textit{observerID} : \textsc{Record} \mapsto \textsc{AgentID}\\
&\textit{reasonerID} : \textsc{Record} \mapsto \textsc{AgentID}\\
&\textit{id} : \textsc{Agent} \mapsto \textsc{AgentID}
\end{aligned}
$$

5.5 Maritime Situation Analysis Framework

The situation analysis framework has a modular design with the minimal yet complete core. The major strength of this design is extensibility. The core of the framework is precise yet flexible to add different modules to it. The creation and refinement of modules can improve the design by adding more details to it, or modifying it for specific purposes.

The core of the situation analysis framework contains two main components *Observer Controller* and *Reasoner Controller*. The figure 5.5 illustrates the flow of data and information in the form of observation segments between the modules of the Observer and the Reasoner. The situation analysis framework is a pipeline which contains different rules in different modules. The output of each rule stores in a set (set of observation segments), thus the next module can use it as its input. The framework is fed by observation from the environment (i.e., raw observation segments) and at the end the global trajectory representations is the output of it.

This proposed framework is built upon the foundation of the framework in [37]. Nalbandyan's [37] framework served as the basis for my working model, after undergoing modifications and enrichments to flesh out an improved framework for maritime situation analysis. In this section each module of the framework (except the Inter-Reasoner modules) is defined precisely and concisely; moreover, they have been validated with ASM.

In the Observer, raw observation segments are cleaned (of outliers) and stored in the Cleaned Observation Segments set. All the cleaned and valid observation segments with similar timestamps related to the same object will be merged (in the time series), to build a generic and observation of the observer (stored in the Merged Observation Segments set). In the Reasoner, all the merged observation segments belonging to each observer will be merged in order to build a time series of each object (stored in the Unified Observation Segments set). All the unified observation segments of all the observers which belong to a same reasoner are the input of the Combine Observations module of that reasoner. Thus, all their corresponding time series (which come from different observers), merge to build a coherent time series for each object (and are stored in the Fused Observation Segments set). After this, all inconsistencies (which is a natural result of having observations from different sensors) will be resolved and the confidence of different sensors are adjusted (Consistent Fused Observation Segments set).

Up to this point, each reasoner has a coherent and consistent time series for each object. As it states in [37], Detect (Object) Relationships is about detecting interactions between two or more objects, utilizing contextual information and other additional sources, which are suspicious and requires special attention or even interjection [42, 43].

As previously noted, the modules within of the Inter-Reasoner are not part the focus of this work. However, the Inter-Reasoner is an integral part of the situation analysis framework, and is explained in [37] as below;

Inter-Reasoner Controller:

- **Exchange Information**: Each reasoner agent communicates with a set of neighbors (i.e., reasoner agents), which may vary from a single one to all the others. If there is any new relevant information to share with neighbors after performing the *Detect Relationships* rule, the agent prepares the new information and communicates it to its neighbors in the logical network.
- **Distributed Agreement**: In every state, reasoner agents assess the newly received information. If this conflicts with their own local "perception" of a situation, they apply conflict resolution strategies using confidence values and eventually agree on the situation. Upon reaching mutual agreement, the updated information is shared with neighbors, eventually forming coherent *global observations*. Different distributed agreement protocols exist in literature, and the selection of a suitable protocol is context dependent.

Since the network structure potentially contains more than one reasoner, the inter-reasoner can hypothetically be assumed as a super-reasoner and it plays the role of the distributed agreement protocol between all reasoners. The input of this super-reasoner is

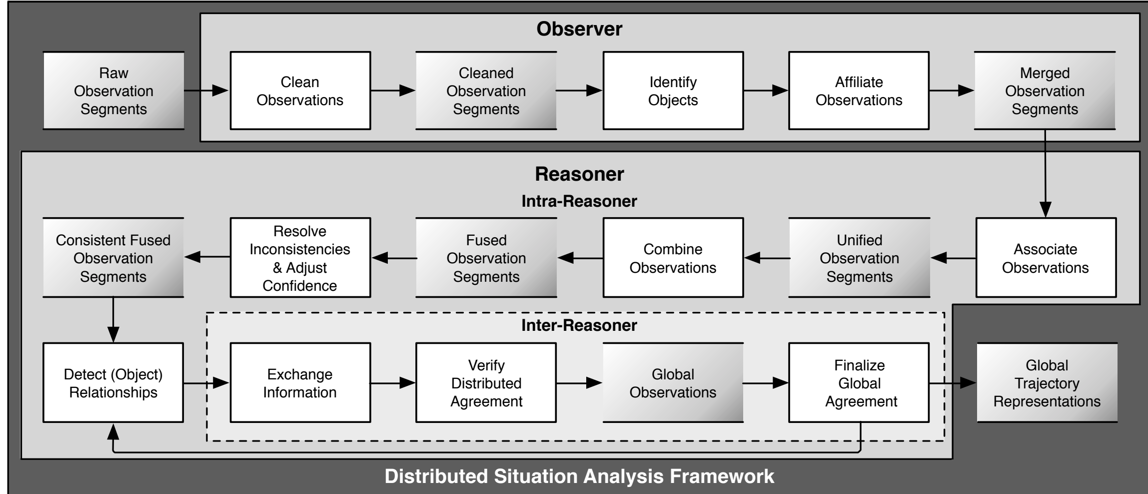


Figure 5.5: Data and Information Flow[3]

the perception of each reasoner from the area which they cover. The output of the supper-reasoner is the global trajectory representations which is the outcome of the agreement between all reasoners. In the last component of the inter-reasoner, Finalize Global Agreement, trajectory of a vessel is being finalized.

// Functions and Rules Related to Observation Segment

$$\begin{aligned}
&\mathit{rawOSs} : \textsc{Sensor} \mapsto \textsc{Set}(\textsc{ObservationSegment})\\
&\mathit{cleanedOSs} : \textsc{Observer} \mapsto \textsc{Set}(\textsc{ObservationSegment})\\
&\mathit{mergedOSs} : \textsc{Observer} \mapsto \textsc{Set}(\textsc{ObservationSegment})\\
&\mathit{unifiedOSs} : \textsc{Reasoner} \mapsto \textsc{Set}(\textsc{ObservationSegment})\\
&\mathit{fusedOSs} : \textsc{Reasoner} \mapsto \textsc{Set}(\textsc{ObservationSegment})\\
&\mathit{cFusedOSs} : \textsc{Reasoner} \mapsto \textsc{Set}(\textsc{ObservationSegment})
\end{aligned}$$

$$\begin{aligned}
&\mathit{observerObjectSet} : \textsc{Observer} \mapsto \textsc{Set}(\textsc{Object})\\
&\mathit{reasonerObjectSet} : \textsc{Reasoner} \mapsto \textsc{Set}(\textsc{Object})\\
&\mathit{cleanedSet} : \textsc{Observer} \times \textsc{Object} \mapsto \textsc{Set}(\textsc{ObservationSegment})\\
&\mathit{unifiedSet} : \textsc{Reasoner} \times \textsc{Object} \mapsto \textsc{Set}(\textsc{ObservationSegment})
\end{aligned}$$

$\textsf{\textbf{IsDefined}}(os : \textsc{ObservationSegment}) \stackrel{\text{def}}{=} os \neq undef$

GenerateAUniversallyRandomID =
  extend $\textsc{Object}$ **with** *tempObj*

[3]This is a revised and extended representation of the model shown in [37]

5.5.1 Observer Controller

An ASM agent is assigned to each observer (based on the network structure) for synchronously running all the rules in the Observer in parallel.

Observer =
  CleanObservations
  IdentifyObjects
  AffiliateObservations

Clean Observations

A specific valid range as explained briefly in Table 2.1 is assigned to each feature in a head (e.g., speed, course, etc.) for any AIS report. The CleanObservations rule identifies and marks the invalid values in raw observation segments which are received from different types of sensors. In addition, the owner of the observation segment is changed to the observer of the sensors based on the network structure. The main assumption of the rule enforces that received raw observation segment from a sensor for one single timestamp is related to a single object. Thus, $|Body(\mathcal{T})| = 1$ and $startTimestamp(\mathcal{T}) = endTimestamp(\mathcal{T})$.

```
CleanObservations =
  forall snr in sensors(self) do
    forall ros in rawOSs(snr) with (status(ros) ≠ cleaned) do
        extend cleanedOSs(self) with cos do
            par
              owner(cos) := self
              object(cos) := object(ros)
              timeSeries(cos) := Clean(timeSeries(ros))
              status(cos) := undef
              status(ros) := cleaned
 where
  Clean(ts : TimeSeries) =
    let timestamp := startTimestamp(ts) in   // startTimestamp(ts) = endTimestamp(ts)
      forall f in head(ts) do                // because |body(ts)| = 1
        forall rec in record(ts, timestamp, f) do
          if (value(rec) < δ_l(f) ∨ value(rec) > δ_u(f)) then
            value(rec) := 'out-of-range'  // Each feature has a specific valid range
```

Identify Object

Each object must be defined with an unique ID to trace. An object may not have an ID, either because the type of sensor is not capable to detect the object's ID (e.g., radar, satellite images) or because of corruption or missing data. In IdentifyObject, the correct ID will be assigned to the correct object if it does not have its own ID assigned.

First Refinement – The GenerateAUniversallyRandomID rule generates a temporary unique ID for any object which is not assigned an ID. The temporary ID helps to keep track of the object and must be different and distinguishable from any actual object's ID.

IdentifyObjects =
  forall *cos* **in** *cleanedOSs*(*self*) **with** (*status*(*cos*) $\neq$ *merged*) **do**
    if (*object*(*cos*) = *undef*) **then**
      GenerateAUniversallyRandomID

Affiliate Observations

For each object, all cleaned observation segments of an observer at a similar timestamp will be merged (horizontally) into one single observation segment to create the generic local observation of that observer. The local observation has all data of different features from different types of sensors (sensors of the observer) at that specific timestamp, Figure 5.6.

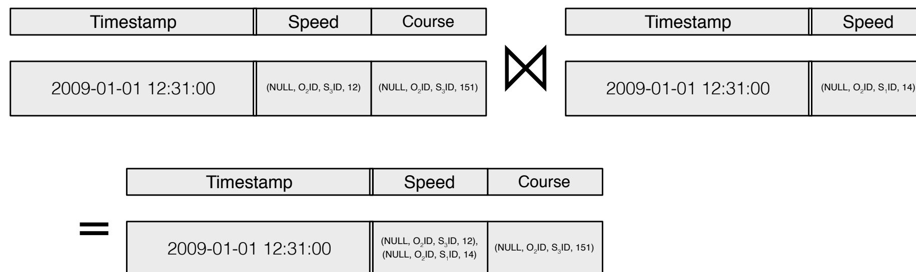


Figure 5.6: Affiliate Observations Rule

derived $observerObjectSet(self) =$
$\{object(cos) \mid cos \in cleanedOSs(self) \wedge status(cos) \neq merged\}$

derived $cleanedSet(self, obj) =$
$\{cos \mid cos \in cleanedOSs(self) \wedge object(cos) = obj \wedge status(cos) \neq merged\}$

AffiliateObservations =
  forall obj **in** $observerObjectSet(self)$ **do**
    extend $mergedOSs(self)$ **with** mos **do** // $|body(mos)| = 1$
      par
        $owner(mos) := self$
        $object(mos) := obj$
        $timeSeries(mos) := \bowtie_{\forall cos\ :\ cos \in cleanedSet(self, obj)} (timeSeries(cos))$
        $status(mos) := undef$
        forall cos **in** $cleanedSet(self, obj)$ **do**
          $status(cos) := merged$

5.5.2 Reasoner Controller

An ASM agent is assigned to each reasoner (based on the network structure) for synchronously running all the rules in the Reasoner in parallel.

Reasoner =
  AssociateObservations
  CombineObservations
  ResolveInconsistencies
  AdjustConfidence

Associate Observations

The AssociateObservations rule is executed by reasoner agents. In this rule, all the merged observation segments which belong to the same observer and same object will be merged (vertically) to create a time series (since they have different timestamps), Figure 5.7. In other words, for each observer, merged observation segments of an object will be associated to a unified observation segment with the similar object to create a more up-to-date time series in that unified observation segment.

It is important to note, these time series are growing within all observers of a reasoner independently; therefore, the AssociateObservations rule can be potentially executed by observer agents; however, due to the following reasons, the AssociateObservations is located in the Reasoner. Time and space limitations at the implementation level enforce that time

series of unified observation segments only grow to a certain point. Thus the time series passed to (CombineObservations) should be partitioned. On the other hand and due to the nature of Observers, they cannot make any decisions on the partitioning process. Therefore the AssociateObservations rule must be located at the Reasoner level.

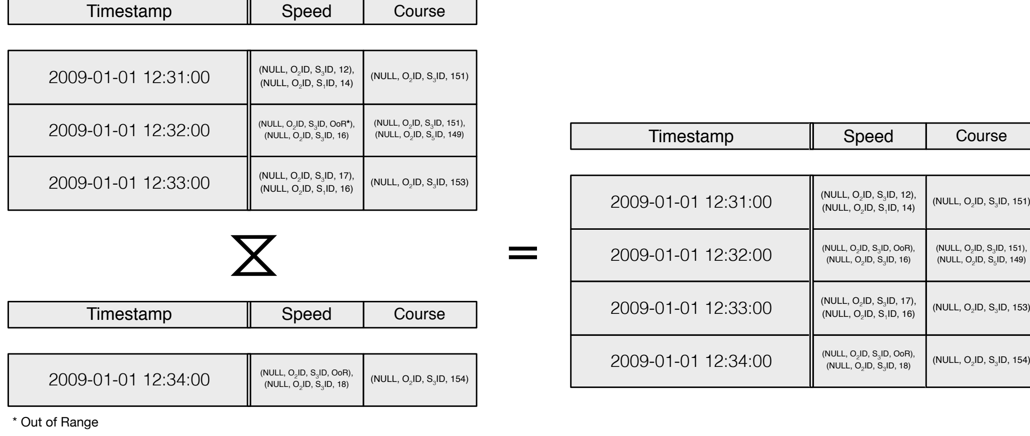


Figure 5.7: Associate Observations Rule

---

derived *observerObjectSet*(*obs*) =
$\{object(mos) \mid mos \in mergedOSs(obs) \wedge status(mos) \neq unified\}$

AssociateObservations =
  forall *obs* **in** *observers*(*self*) **do**
    forall *obj* **in** *observerObjectSet*(*obs*) **do**
      choose *mos* **in** *mergedOSs*(*obs*) **with** $(object(mos) = obj \wedge status(mos) \neq unified)$ **do**
        choose *uos* **in** *unifiedOSs*(*obs*) **with** $(object(uos) = obj)$ **do**
          if IsDefined(*uos*) **then**
            par
              $timeSeries(uos) := ⧖(timeSeries(uos), timeSeries(mos))$
              $status(uos) := undef$
              $status(mos) := unified$
          else
            extend *unifiedOSs*(*self*) **with** *newUos* **do**
              par
                $owner(newUos) := self$
                $object(newUos) := obj$
                $timeSeries(newUos) := timeSeries(mos)$
                $status(newUos) := undef$
                $status(mos) := unified$

Combine Observations

To have the most coherent picture of an object's movement in the real-world, from the view of a reasoner, all the unified observation segments of each observer (which belong to that reasoner based on the network structure) should be merged. Since all the unified observation segments of an object are from different observers with different sets of sensors; the head of the time series of unified observation segments may not be similar. On the other hand, time series of different unified observation segments have different timestamps as explained in Section 5.5.2.Associate Observations. As a result, merging in CombineObservations is a hybrid merge.

---

derived *reasonerObjectSet*(*self*) =

$$\bigcup_{\forall obs\ :\ obs \in observers(self)} \{object(uos) \mid uos \in unifiedOSs(obs) \wedge status(uos) \neq fused\}$$

derived *unifiedSet*(*self*, *obj*) =

$$\bigcup_{\forall obs\ :\ obs \in observers(self)} \{uos \mid uos \in unifiedOSs(obs) \wedge object(uos) = obj \wedge status(uos) \neq fused\}$$

CombineObservations =
  forall *obj* **in** *reasonerObjectSet*(*self*) **do**
    choose *fos* **in** *fusedOSs*(*self*) **with** (*object*(*fos*) = *obj*) **do**
      if IsDefined(*fos*) **then**
        par
          $timeSeries(fos) := \boxtimes_{\forall uos\ :\ uos \in unifiedSet(self, obj)} (timeSeries(fos), timeSeries(uos))$
          status(*fos*) := *undef*
          forall *uos* **in** *unifiedSet*(*self*, *obj*) **do**
            status(*uos*) := *fused*
      else
        extend *fusedOSs*(*self*) **with** *newFos* **do**
          par
            owner(*newFos*) := *self*
            object(*newFos*) := *object*(*uos*)
            $timeSeries(newFos) := \boxtimes_{\forall uos\ :\ uos \in unifiedSet(self, obj)} (timeSeries(uos))$
            status(*newFos*) := *undef*
            forall *uos* **in** *unifiedSet*(*self*, *obj*) **do**
              status(*uos*) := *fused*

Resolve Inconsistencies

For each feature in a tuple there may be more than one record which contains different values (due to the first refinement on merging time series). ResolveInconsistencies attempts to condense to one single record based on all the records.

Resolve Inconsistencies – Primitive Solution

The primitive solution would be that the Resolve rule chooses one record at random. This approach of resolve inconsistencies is not involved with sensor confidence, nor sensor's errors of the received data.

```
ResolveInconsistencies =
  forall fos in fusedOSs(self) with (status(fos) ≠ consistent) do
    extend cFusedOSs(self) with cFos do
       par
         owner(cFos) := self
         object(cFos) := object(fos)
         timeSeries(cFos) := ConflictResolution(timeSeries(fos))
         status(cFos) := undef
         status(fos) := consistent
 where
  ConflictResolution(ts : TimeSeries) =
    forall f in head(ts) do
      forall t in body(ts) do
        let recordSet := record(ts, tupleTimestamp(t), f) in
          extend recordSet with resolvedRecord do
            if |recordSet| = 0 then
              Nullify(recordSet, resolvedRecord)
            else if |recordSet| = 1 then
              Confirm(recordSet, resolvedRecord)
            else
              Resolve(recordSet, resolvedRecord)

    Nullify(recordSet : Set(Record), resolvedRecord : Record) =
      value(resolvedRecord) := null
      sensorID(resolvedRecord) := null
      observerID(resolvedRecord) := null
      reasonerID(resolvedRecord) := id(self)

    Confirm(recordSet : Set(Record), resolvedRecord : Record) =
      choose rec in recordSet do
        value(resolvedRecord) := value(rec)
        sensorID(resolvedRecord) := sensorID(rec)
        observerID(resolvedRecord) := observerID(rec)
        reasonerID(resolvedRecord) := id(self)

    Resolve(recordSet : Set(Record), resolvedRecord : Record) =
      choose rec in recordSet do                 // One of the records will be chosen randomly
        value(resolvedRecord) := value(rec)      //  and no confidence is applied
        sensorID(resolvedRecord) := sensorID(rec)
        observerID(resolvedRecord) := observerID(rec)
        reasonerID(resolvedRecord) := id(self)
```

Resolve Inconsistencies – Considering Sensors' Error

In observational studies, repeated measurement is the key to obtaining as close a value as possible to the actual value of a variable. Calculating the uncertainty of measurements because of the error of the measuring device (or devices) is another concern which statistics helps to conquer. However, finding a closer value to the actual value and an error of this value can be done in various ways as explained later.

In the framework, Figure 5.5, the rules ResolveInconsistencies and AdjustConfidence will be accurate only if the definition of confidence is clear and precise.

Definition – Confidence

Confidence of a sensor is defined as the error of that sensor, ΔS_i, which is not determined by each individual observation. It is a value which is derived of how accurate a sensor is over time and may change with an external agent. If the error of a sensor becomes greater than a defined threshold, the received value of that sensor will not be considered until it becomes reliable again.

In this refinement, the structure of each record will be slightly different. Each record will carry the error of its own sensor (or the error of number of sensors after AdjustConfidence); therefore, *(ReasonerID ,ObserverID ,SensorID, Error, Value).*

It is important to note that the structure of the ResolveInconsistencies rule will not change, thus it contains the Nullify rule, the Confirm rule, and the Resolve rule. The refinement will be mainly on the Resolve rule which applies when there is more than one record in a cell.

Calculating Value – Avarage

The Resolve rule should condense to a single record, which is the simple average of all the records' value in this refinement as below;

$$v_{R_n} = z = \frac{v_{S_i} \ + \ v_{S_{i+1}} \ + \ \ ... \ \ + \ v_{S_{i+k}}}{k+1}$$

– v_{S_i} is the value of a record which was received from sensor S_i.

– $v_{S_i}, v_{S_{i+1}}, \ldots, v_{S_k}$ are all in a same cell of a merged time series.

– $S_i, S_{i+1}, \ldots, S_k$ are the sensors belonging to a reasoner R_n (They may or may not belong to the same observer).

Adjust Confidence

Repeated measurement can be done in different ways such as one device measures several times, or a number of independent devices measure. In this case, different sensors measure

a feature of an object independently and eventually all data is merge. As a result, there may be more than one measurement (i.e., record) in a cell of a time series. The output of the rule Resolve is a single value based on the values of all records (in this refinement it is the average of them). AdjustConfidence rule calculates the error of the output record in Resolve (based on the error of each record in the cell).

Calculating Error – Propagation of Uncertainty

In statistics, if the measurements are independent (therefore, the errors are independent as well), the error of their average (or in general, any other function) will be calculated as below;

$$\Delta R_n = \sqrt{(\frac{\partial z}{\partial v_{S_i}})^2.\Delta {S_i}^2 + (\frac{\partial z}{\partial v_{S_{i+1}}})^2.\Delta {S_{i+1}}^2 + \ldots + (\frac{\partial z}{\partial v_{S_{i+k}}})^2.\Delta {S_{i+k}}^2}$$

Note that, reasoner themselves do not have error, what it considered as reasoner's error, ΔR_n, is the calculated error of the records in a cell of a merged time series. For instance, if two records such as $(RID_n, OID_m, SID_i, \Delta S_i, V_{S_i})$ and $(RID_n, OID_{m'}, SID_j, \Delta S_j, V_{S_j})$, where S_i and S_j are the sensors of similar or different observers, are in the same cell, the output of ResolveInconsistencies and AdjustConfidence rules will be a record such as $(RID_n, Null, Null, \Delta R_n, V_{average})$. Therefore ΔR_n is the calculated error based on ΔS_i and ΔS_j.

As a result of this approach, in each cell of time series, the error of a reasoner is calculating; therefore, there might be different errors for one reasoner. The average of all the errors for each reasoner is the error of that reasoner.

// Functions and Rules Related to the First Refinement

error : RECORD $\mapsto$ ERROR
valuesOfRecordSet : SET(RECORD) $\mapsto$ SET(NUMBER)
errorsOfRecordSet : SET(RECORD) $\mapsto$ SET(NUMBER)

Average(SET(NUMBER))
Error(SET(NUMBER))

derived $valuesOfRecordSet(self) = \{value(rec) \mid rec \in recordSet\}$
derived $errorsOfRecordSet(self) = \{error(rec) \mid rec \in recordSet\}$

Nullify$(recordSet : \textsc{Set}(\textsc{Record}), resolvedRecord : \textsc{Record}) =$
$value(resolvedRecord) := null$
$error(resolvedRecord) := null$
$sensorID(resolvedRecord) := null$
$observerID(resolvedRecord) := id(self)$
$reasonerID(resolvedRecord) := null$

Confirm$(recordSet : \textsc{Set}(\textsc{Record}), resolvedRecord : \textsc{Record}) =$
choose rec **in** $recordSet$ **do**
$value(resolvedRecord) := value(rec)$
$error(resolvedRecord) := error(rec)$
$sensorID(resolvedRecord) := sensorID(rec)$
$observerID(resolvedRecord) := observerID(rec)$
$reasonerID(resolvedRecord) := id(self)$

Resolve$(recordSet : \textsc{Set}(\textsc{Record}), resolvedRecord : \textsc{Record}) =$
$value(resolvedRecord) := \mathsf{Average}(valuesOfRecordSet)$
$error(resolvedRecord) := \mathsf{Error}(errorsOfRecordSet)$
$sensorID(resolvedRecord) := null$
$observerID(resolvedRecord) := null$ // If all of the observerIDs in a *recordSet* are similar,
$reasonerID(resolvedRecord) := id(self)$ // observerID(resolvedRecord) := *observerID*(rec)

5.6 Formal Definition of Trajectory

Global Trajectory Representations are a set of observation segments, each of which is related to one object. The time series of each observation segment is called the *trajectory* of the *object* of that observation segment. Trajectory of an object is the most general outcome (of the framework based on all the reasoners) which represents the object in the real-world situation. Based on the definition of trajectory in [37], we present here a refined version to make the definition consistent with the definition of observation segment and time series. We also provide a proof of this critical concept, which is not given in [37].

$trajectory : \textsc{Object} \mapsto \textsc{TimeSeries}$
derived $trajectory(obj) \equiv \hat{trajectory}(obj,\ currentTimestamp)$ // which *currentTimestamp*
// is the latest timestamp of
// time series (related to *obj*)
// in Global Trajectory Representations

where the inductive definition of *trajectory* is as follows:
$\hat{trajectory} : \textsc{Object} \times \textsc{Time} \mapsto \textsc{TimeSeries}$
$\hat{trajectory}(obj,\ currentTimestamp) \stackrel{\text{def}}{=}$
$\hat{trajectory}(obj,\ previousTimestamp) \boxtimes \{timeSeries(OS) \mid OS \in GlobalTrajectoryRepresentations \wedge endTimestamp(timeSeries(OS)) = currentTimestamp \wedge object(OS) = obj\}$ // which *previousTimestamp*
// precedes *currentTimestamp*
$\hat{trajectory}(obj, beginningTimestamp) := \emptyset$ // which *beginningTimestamp* is
// when the model started to run

Corollary 1. The function trajectory is a bijective mapping between the set of objects in any given state and their history of global trajectory representations.

To prove the function trajectory is bijective, we need to prove trajectory is *injective* and *surjective*.
Injectivity: The trajectory function being an injective function means no two elements in the domain of the function (i.e., objects) get mapped to the same image (i.e., time series).

$\forall obj, obj' \in \textsc{Object}$ which $obj \neq obj' \implies trajectory(obj) \neq trajectory(obj')$
where $\textsc{Object} = \{object(OS) \mid OS \in GlobalTrajectoryRepresentations\}$

Proof : Suppose this proposition is false.
This conditional statement being false means there are two different objects (assume *obj* and obj') which implies $trajectory(obj) = trajectory(obj')$ thus, based on the definition of $\hat{trajectory}$;

$$\hat{trajectory}(obj,\ currentTimestamp) = \hat{trajectory}(obj',\ currentTimestamp')$$

If $currentTimestamp \neq currentTimestamp'$, then $trajectory(obj) \neq trajectory(obj')$ and it is a contradiction; therefore, the proposition is true.

If $currentTimestamp = currentTimestamp'$, then

$$\begin{aligned}\hat{trajectory}(obj,\ currentTimestamp) = \hat{trajectory}(obj,\ previousTimestamp)\ \boxtimes \\ \{timeSeries(OS) \mid OS \in GlobalTrajectoryRepresentations\ \wedge \\ endTimestamp(timeSeries(OS)) = currentTimestamp\ \wedge \\ object(OS) = obj\}\end{aligned}$$

$$\begin{aligned}\hat{trajectory}(obj',\ currentTimestamp) = \hat{trajectory}(obj',\ previousTimestamp')\ \boxtimes \\ \{timeSeries(OS) \mid OS \in GlobalTrajectoryRepresentations\ \wedge \\ endTimestamp(timeSeries(OS)) = currentTimestamp\ \wedge \\ object(OS) = obj'\}\end{aligned}$$

Even if the time series of two objects, obj and obj' have the same timestamp, *currentTimestamp*, they are different time series with different values in their body, because they are related to two different objects. Therefore, $trajectory(obj) \neq trajectory(obj')$ and it is a contradiction and it means the proposition is true and trajectory is injective.

Surjectivity: The trajectory function being a surjective function means any element in the range of the function is hit by the function.

Proof : Based on the definition of the domain and the range of the trajectory function, surjectivity is intuitive.

$$\textsc{Object} = \{object(OS) \mid OS \in GlobalTrajectoryRepresentations\}$$
$$\textsc{TimeSeries} = \{timeSeries(OS) \mid OS \in GlobalTrajectoryRepresentations\}$$

Any element in the range of trajectory is a time series of an observation segment in the global trajectory representations, which is mapped to the object of same observation segment in the domain of trajectory; therefore trajectory is surjective.

Corollary 2. For each object *obj* in any state of model, $trajectory(obj)$ is a time series of coherent and, in a probabilistic sense based on confidence values, consistent observations of *obj* over a run of the SA model.[4]

[4]Intuitively, the function *trajectory* yields the best possible situational evidence one can compute under uncertainty from multiple observations of how objects move.

Chapter 6

Implementation

CoreASM [15, 17, 19] is an Open Source project and a supporting tool to focus on early phases of the software design process. The main goal of CoreASM is encouraging rapid prototyping, analyzing, as well as experimental validation of ASM models. CoreASM has an extensible plugin-based architecture [18] and the core of the language and engine (i.e., *kernel*) holds only to the bare essentials. Most of the language constructs and functions are defined in plugins as extensions to the kernel. CoreASM plugins can either extend the functionality of specific engine components (introducing additional structure or behavior to those components), or they can extend the control flow of the engine (interposing their own code in between state transitions of the engine). Extending with plugins gives a better control on CoreASM, allowing for customizing depending on specific application needs. CoreASM is implemented in Java under Academic Free License version 3.0 (AFL 3.0) and it is available at `www.coreasm.org`.

The proposed situation analysis framework is implemented using CoreASM and Java. The part which is implemented in CoreASM has a higher level of abstraction and its responsibility is to have control on time, input/output (calling the Java method for each sensor to read the raw data from a text file); moreover, CoreASM has access to the network structure; therefore it is calling all the related method/functions of Java for each rule in Observers and Reasoners.

On the other hand, Java mainly deals with data structures and the actual data. Data structure for observation segment, time series, tuple, and record are defined in Java; moreover, all of the merges (i.e., horizontal merge and vertical merge) are defined and implemented as methods in Java as well. Rules are implemented in Java from reading raw data from a text file, to every rule in Observer and Reasoner, thus, when CoreASM calls these rules, they can be executed for a specific sensor, observer, or reasoner.

6.1 Implementation of the Framework (CoreASM Part)

For each physical sensor in the environment there is a corresponding agent in CoreASM. For each sensor's agent, CoreASM calls the related method in Java for reading raw data, which is stored in a text file with a specific format and name. Moreover, for each observer and reasoner there is a corresponding agent defined in CoreASM which is responsible for calling all the related method/functions in Java. JASMine is a CoreASM plugin which provides the access to Java objects and classes from CoreASM [22]. At first, CoreASM imports a class into a location and then invokes a method of that class from Java. Furthermore, to increase the readability of the code, Modularity plugin helps breaking the code into different CoreASM modules as explained below.

6.1.1 Network Configuration Module

NetworkConfiguration is the rule of Network Configuration module in CoreASM which executes before any other rule in the Initialization rule. The hierarchical structure of the network is building by executing this rule (Figure 5.3). Based on four ASM functions below, each sensor belongs to one observer (*observer* function), and each observer belongs to one reasoner (*reasoner* function). On the other hand, the sensor set of each observer and the observer set of each reasoner establishes in the NetworkConfiguration as well (*sensors* function and *observers* function respectively).

```
function reasoner: OBSERVER -> REASONER
function observers: REASONER -> SET //SET OF OBSERVER
function observer: SENSOR -> OBSERVER
function sensors: OBSERVER -> SET //SET OF SENSOR
```

Furthermore, in this module each AgentID is assigned to the relevant agent (i.e., all sensors, observers, and reasoners).

```
function id: AGENT -> AGENTID //AGENT = UNION(SENSOR, OBERVER, REASONER)
```

6.1.2 Environment Module

In our implementation (in CoreASM), we define the concept of "time" as logical time. This means that "time" is set to *zero* in the initial state; and then, the Environment rule is responsible for increasing it by *one* unit at every step.
The Environment rule invokes the *readFromCSVFile* method for all active sensors (i.e., sensor agent). *readFromCSVFile* is a Java method which reads input values from a CSV[1] file. It has two input parameters, the sensor name and the logical time.

[1]Comma-Separated Values

Due to the fact that the accuracy of the measurement of timestamps is one second in the real-world, converting a logical time to a real-world timestamps in Java (the logical time will be an parameter of a Java method) is a viable solution to keep the CoreASM part as a controller in the abstract level.

```
forall snr in SENSOR with status(snr) = Enable do
   invoke environment -> readFromCSVFile(id(snr), time)
```

6.1.3 Observer and Reasoner Modules

There are two separate modules related to the Observer and the Reasoner. As stated in Section 5.5, each observer agent runs all of the sub-rules in Observer in parallel. Likewise, each reasoner agent runs all of the sub-rules in Reasoner in parallel.

Observer Rule

```
rule Observer = {
   CleanObservations
   IdentifyObjects
   AffiliateObservations
}

rule CleanObservations = {
   forall snr in sensors(self) do
      invoke coreASMCleanObservations -> cleanedOSs(id(self), id(snr))
}

rule AffiliateObservations = {
   invoke coreASMAffiliateObservations -> MergedOSs(id(self))
}
```

Reasoner Rule

```
rule Reasoner = {
   AssociateObservations
   CombineObservations
   ResolveInconsistenciesAndAdjustConfidence
}

rule AssociateObservations = {
   forall obs in observers(self) do
      invoke coreASMAssociateObservations -> UnifiedOSs(id(self), id(obs))
}

rule CombineObservations = {
   invoke coreASMCombineObservations -> FusedOSs(id(self))
```

```
}

rule ResolveInconsistenciesAndAdjustConfidence = {
   invoke coreASMResolveInconsistenciesAndAdjustConfidence -> ConsistentFusedOSs(id(self))
}
```

6.1.4 Situation Analysis (SA) Module

SA is the main module in our implementation in CoreASM, as such the init rule executes here. All functions, universes, and enums are defined in SA. As previously mentioned, agents are defined and assigned in this module as follows;

- There is an agent for each observer and reasoner.
- There is an agent, called EnvironmentAgent, which is responsible for executing the Environment rule.

Different *modes* are defined in this module to put order in running different rules. Initially, when the mode is *undef*, the logical time is set to *zero* and the NetworkConfiguration rule is being executed. All Java classes are imported in *importJavaFiles* mode. When the mode is changed to *generateOSsStructure*, all data structures are created by invoking related methods from Java for each sensor, observer, and reasoner. Finally, all observers, reasoners, and environment agents are assigned to the Observer, Reasoner, and Environment rules, respectively.

```
init Initialization

rule Initialization = {
   if (mode = undef) then {
      time := 0

      NetworkConfiguration

      mode := importJavaFiles
   }

   else if (mode = importJavaFiles) then {
      import native io.IOManager() into environment

      import native coreasm.Initialization() into initialization

      import native coreasm.CleanObservations() into coreASMCleanObservations
      import native coreasm.AffiliateObservations() into coreASMAffiliateObservations

      import native coreasm.AssociateObservations() into coreASMAssociateObservations
      import native coreasm.CombineObservations() into coreASMCombineObservations
```

```
    import native coreasm.ResolveInconsistencies() into
        coreASMResolveInconsistenciesAndAdjustConfidence

    import native core.PrintOSs() into coreASMPrintOSs

    mode := generateOSsStructure
  }

  else if (mode = generateOSsStructure) then {
    forall snr in SENSOR with status(snr) = Enable do
      invoke initialization -> generateSensorOSsStructure(id(snr))

    forall obs in OBSERVER do
      invoke initialization -> generateObserverOSsStructure(id(obs))

    forall rsn in REASONER do
      invoke initialization -> generateReasonerOSsStructure(id(rsn))

    mode := assignAgents
  }

  else if (mode = assignAgents) then {
    program(EnvironmentAgent) := @Environment

    forall obs in OBSERVER do {
      Agents(obs) := true
      program(obs) := @Observer
    }

    forall rsn in REASONER do {
      Agents(rsn) := true
      program(rsn) := @Reasoner
    }

    program(self) := undef
  }

}
```

6.2 Implementation of the Framework (Java Part)

CoreASM is in charge of controlling the logical time as well as the Environment, Observer and Reasoner rules. Each of which has a corresponding method in Java with at least one parameter. This parameter is the ID of the agent (e.g., sensor, observer, or reasoner agent) and is passed to Java by CoreASM. Records, tuples, and time series structures are defined as

separate classes in Java. Also, an observation segment class is defined in Java to support the data structure corresponding to raw observation segments, cleaned observation segments, merged observation segments, unified observation segments, fused observation segments, and consistent fused observation segments. Figure 6.1 illustrates the class diagram of domain concepts such as record, time series, observation segment, etc.

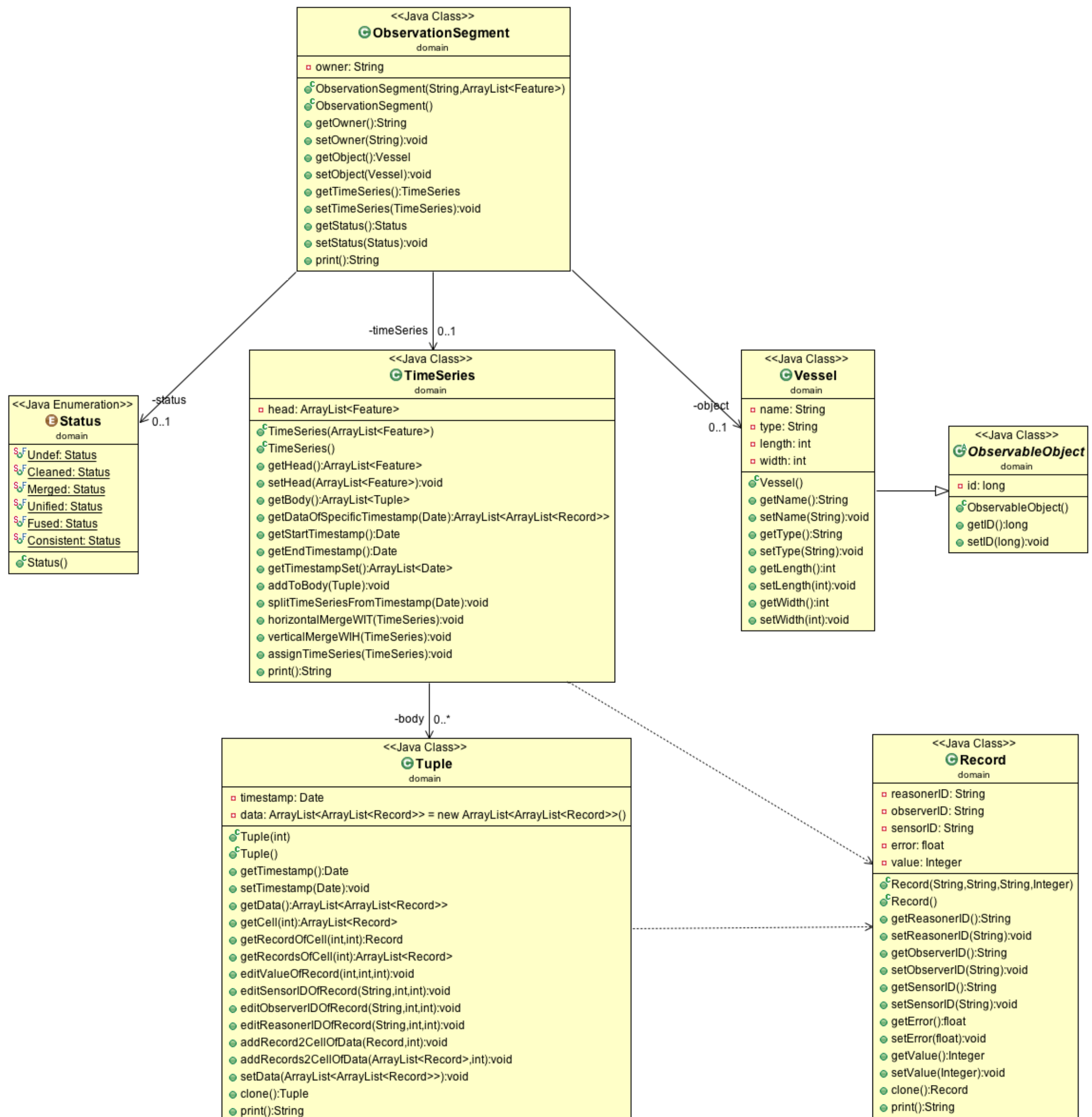


Figure 6.1: Class Diagram of Domain Concepts

6.2.1 Data Structure

Data Structure of Observation Segment

As stated in Section 5.5, observation segments of a specific object, which belong to a observer or a reasoner, are merged in different modules. With this in mind, HashMap is a reasonable choice to store the observation segments as the complexity of accessing to an observation segment of a certain object which belongs to an observer/reasoner is of $O(1)$.

As illustrated in Figure 6.2, any of the Cleaned Observation Segments, Merged Observation Segments, Fused Observation Segments, and Consistent Fused Observation Segments requires a HashMap of HashMaps [observer/reasoner , [object , observation segment]].

Only the Unified Observation Segments requires a three-layered HashMap due to the fact that the owner of observation segments in Unified Observation Segments is a reasoner, but observation segments of a similar observer for a specific object are getting merged. So, we need to have all reasoners, all of the corresponding observers, and all the objects in one structure.

These structures are defined as a set of public static members in a class named Globals as below;

```
public static HashMap<String, HashMap<Long, ArrayList<ObservationSegment>>> rawOSs =
  new HashMap<String, HashMap<Long, ArrayList<ObservationSegment>>>();

public static HashMap<String, HashMap<Long, ArrayList<ObservationSegment>>> cleanedOSs =
  new HashMap<String, HashMap<Long, ArrayList<ObservationSegment>>>();
public static HashMap<String, HashMap<Long, ArrayList<ObservationSegment>>> mergedOSs =
  new HashMap<String, HashMap<Long, ArrayList<ObservationSegment>>>();

public static HashMap<String, HashMap<String, HashMap<Long,
    ArrayList<ObservationSegment>>>> unifiedOSs =
  new HashMap<String, HashMap<String, HashMap<Long, ArrayList<ObservationSegment>>>>();
public static HashMap<String, HashMap<Long, ObservationSegment>> fusedOSs =
  new HashMap<String, HashMap<Long, ObservationSegment>>();
public static HashMap<String, HashMap<Long, ObservationSegment>> cfusedOSs =
  new HashMap<String, HashMap<Long, ObservationSegment>>();
```

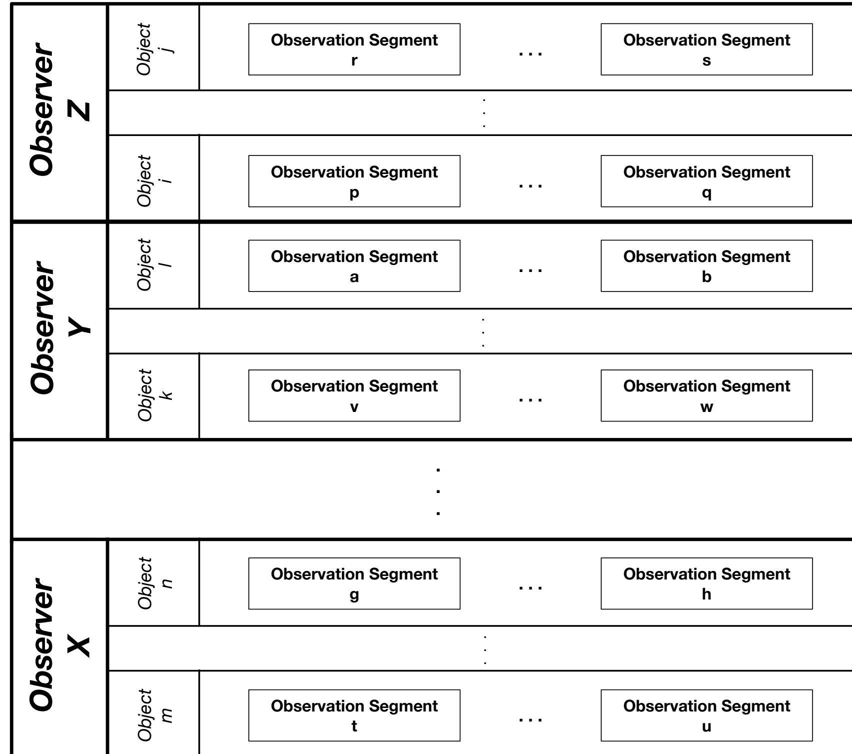


Figure 6.2: Data Structure of Observation Segments with Two-Layer HashMaps

Data Structure of Time Series

Time Series is composed of an ArrayList of Feature as its head, and an ArrayList of Tuple as its body. Feature is a public enum which contains list of used features in our implementation.

```
public class TimeSeries {
  private ArrayList<Feature> head;
  private ArrayList<Tuple> body;
```

Tuple contains a Date and a ArrayList of ArrayLists of records. Note that the size of 'data' (defined below) is the same as the size of the head of time series.

```
public class Tuple {
  private Date timestamp;
  private ArrayList<ArrayList<Record>> data = new ArrayList<ArrayList<Record>>();
```

6.2.2 Implementation of Rules

Each rule of Observer or Reasoner in CoreASM has a corresponding class in Java with a similar name. A simple example, the AffiliateObservations rule, is presented in the following and the rest of the implementation details are presented in Appendix A.

```java
public class AffiliateObservations {

   public void MergedOSs(String observerID) throws IOException {
      if (!Globals.cleanedOSs.containsKey(observerID))
         throw new IOException("Affiliate Observations Error: The Observer ID, " +
             observerID + ", is NOT valid.");
      if (!Globals.mergedOSs.containsKey(observerID))
         throw new IOException("Affiliate Observations Error: The Observer ID, " +
             observerID + ", is NOT valid.");

      if (!(Globals.cleanedOSs.get(observerID).isEmpty())) {
         HashSet<Long> observerObjectSet = new HashSet<Long>();
         observerObjectSet.addAll(Globals.cleanedOSs.get(observerID).keySet());

         for (Long object : observerObjectSet) {
            for (ObservationSegment cos : Globals.cleanedOSs.get(observerID).get(object)) {
               if (!(cos.getStatus().equals(Status.Merged))) {
                  ObservationSegment mos = new ObservationSegment();

                  mos.setOwner(observerID);
                  mos.setObject(cos.getObject());

                  TimeSeries tempCosTimeseries = new TimeSeries();
                  tempCosTimeseries.assignTimeSeries(cos.getTimeSeries());

                  for (ObservationSegment mergingCos :
                      Globals.cleanedOSs.get(observerID).get(object)) {
                     if ((!(mergingCos.equals(cos))) &&
                         (!(mergingCos.getStatus().equals(Status.Merged))) &&
                        (mergingCos.getTimeSeries().getTimestampSet()
                        .equals(tempCosTimeseries.getTimestampSet()))) {

                        TimeSeries tempMergingCosTimeseries = new TimeSeries();
                        tempMergingCosTimeseries.assignTimeSeries(mergingCos.getTimeSeries());

                        tempCosTimeseries.horizontalMergeWIT(tempMergingCosTimeseries);

                        mergingCos.setStatus(Status.Merged);
                     }
                  }
                  mos.setTimeSeries(tempCosTimeseries);

                  mos.setStatus(Status.Undef);
                  cos.setStatus(Status.Merged);

                  if (Globals.mergedOSs.get(observerID).containsKey(object))
                     Globals.mergedOSs.get(observerID).get(object).add(mos);
```

```
                else {
                    ArrayList<ObservationSegment> mosList = new
                        ArrayList<ObservationSegment>();
                    mosList.add(mos);

                    Globals.mergedOSs.get(observerID).put(object, mosList);
                }
            }
        }
      }
    }
  }

}
```

6.2.3 Empirical Results

In this section, we extract some test data from real-world AIS datasets collected by U.S. Coast Guard Services (available at `www.marinecadastre.gov`) to experimentally validate the implementation. The datasets contain records of marine traffic for each month of the calendar years 2009-2011. Records are filtered to one minute and stored in geodatabases organized by Universal Transverse Mercator (UTM) zones 1-11 and 14-19. Combined, these cover the coastal waters of the entire United States and most of Canada. For our experiments, we assumed there are seven sensors in the environment, as well as four observers and two reasoners. The network configuration is as following;

```
rule NetworkConfiguration = {
  sensors(Observer1) := {Sensor1, Sensor2}
  sensors(Observer2) := {Sensor3, Sensor4}
  sensors(Observer3) := {Sensor5, Sensor6}
  sensors(Observer4) := {Sensor7}

  observers(Reasoner1) := {Observer1, Observer2}
  observers(Reasoner2) := {Observer3, Observer4}
}
```

Input Data From Seven Sensors & Output Time Series

sensor1:

```
Timestamp,MMSI,COG,Heading
2009-01-01 00:00:01,367409990,245,511
2009-01-01 00:00:02,367409990,228,511
2009-01-01 00:00:02,538003423,68,0
```

```
2009-01-01 00:00:03,367409990,235,511
2009-01-01 00:00:04,367409990,248,114
2009-01-01 00:00:04,538003423,74,511
2009-01-01 00:00:05,367409990,258,98
2009-01-01 00:00:06,538003423,69,12
2009-01-01 00:00:07,664445000,163,334
2009-01-01 00:00:08,538003423,84,511
2009-01-01 00:00:09,367409990,250,93
2009-01-01 00:00:10,367409990,261,511
2009-01-01 00:00:10,664445000,172,365
```

---

sensor2:

```
Timestamp,MMSI,SOG,Heading,ROT
2009-01-01 00:00:01,367409990,0,511,128
2009-01-01 00:00:02,367409990,0,125,128
2009-01-01 00:00:02,538003423,12,12,126
2009-01-01 00:00:03,367409990,0,126,127
2009-01-01 00:00:04,367409990,2,129,128
2009-01-01 00:00:04,538003423,15,511,128
2009-01-01 00:00:05,367409990,3,511,126
2009-01-01 00:00:06,538003423,17,511,128
2009-01-01 00:00:07,664445000,16,320,128
2009-01-01 00:00:08,538003423,19,14,124
2009-01-01 00:00:09,367409990,4,139,128
2009-01-01 00:00:10,367409990,4,142,128
2009-01-01 00:00:10,664445000,18,341,121
```

---

sensor3:

```
Timestamp,MMSI,SOG,COG,ROT
2009-01-01 00:00:01,538003423,11,57,127
2009-01-01 00:00:03,367409990,0,240,128
2009-01-01 00:00:03,538003423,14,65,110
2009-01-01 00:00:05,367409990,3,211,128
2009-01-01 00:00:05,538003423,15,71,121
2009-01-01 00:00:06,367409990,3,225,121
2009-01-01 00:00:06,664445000,19,121,127
2009-01-01 00:00:07,367409990,4,213,126
```

```
2009-01-01 00:00:07,538003423,17,84,127
2009-01-01 00:00:07,664445000,21,182,128
2009-01-01 00:00:08,367409990,4,254,128
2009-01-01 00:00:08,664445000,21,175,126
2009-01-01 00:00:09,664445000,23,169,127
2009-01-01 00:00:10,538003423,19,89,120
```

---

sensor4:

```
Timestamp,MMSI,COG,Heading,ROT
2009-01-01 00:00:01,538003423,67,12,126
2009-01-01 00:00:03,367409990,230,511,128
2009-01-01 00:00:03,538003423,73,19,128
2009-01-01 00:00:05,367409990,238,131,127
2009-01-01 00:00:05,538003423,64,511,128
2009-01-01 00:00:06,367409990,234,152,127
2009-01-01 00:00:06,664445000,157,324,128
2009-01-01 00:00:07,367409990,290,511,126
2009-01-01 00:00:07,538003423,81,511,128
2009-01-01 00:00:07,664445000,142,335,128
2009-01-01 00:00:08,367409990,295,167,124
2009-01-01 00:00:08,664445000,159,356,126
2009-01-01 00:00:09,664445000,189,385,128
2009-01-01 00:00:10,538003423,87,21,128
```

---

sensor5:

```
Timestamp,MMSI,SOG,COG,Heading
2009-01-01 00:00:02,367409990,1,210,133
2009-01-01 00:00:02,538003423,14,65,113
2009-01-01 00:00:02,664445000,15,119,313
2009-01-01 00:00:04,367409990,2,237,132
2009-01-01 00:00:04,538003423,15,71,123
2009-01-01 00:00:04,664445000,15,124,334
2009-01-01 00:00:06,367409990,4,247,511
2009-01-01 00:00:06,538003423,17,65,156
2009-01-01 00:00:06,664445000,17,137,365
2009-01-01 00:00:08,367409990,5,295,511
2009-01-01 00:00:08,538003423,17,71,114
```

```
2009-01-01 00:00:08,664445000,21,119,511
2009-01-01 00:00:09,367409990,5,287,116
2009-01-01 00:00:09,538003423,18,91,110
2009-01-01 00:00:09,664445000,23,153,511
2009-01-01 00:00:10,367409990,6,265,118
2009-01-01 00:00:10,538003423,19,86,121
2009-01-01 00:00:10,664445000,24,161,312
```

---

sensor6:

```
Timestamp,MMSI,COG,ROT
2009-01-01 00:00:02,367409990,212,128
2009-01-01 00:00:02,538003423,69,126
2009-01-01 00:00:02,664445000,116,128
2009-01-01 00:00:04,367409990,234,127
2009-01-01 00:00:04,538003423,77,128
2009-01-01 00:00:04,664445000,121,128
2009-01-01 00:00:06,367409990,225,127
2009-01-01 00:00:06,538003423,71,128
2009-01-01 00:00:06,664445000,132,128
2009-01-01 00:00:08,367409990,254,123
2009-01-01 00:00:08,538003423,79,128
2009-01-01 00:00:08,664445000,139,128
2009-01-01 00:00:09,367409990,235,123
2009-01-01 00:00:09,538003423,91,128
2009-01-01 00:00:09,664445000,140,125
2009-01-01 00:00:10,367409990,264,127
2009-01-01 00:00:10,538003423,98,128
2009-01-01 00:00:10,664445000,153,128
```

---

sensor7:

```
Timestamp,MMSI,SOG,COG,Heading,ROT
2009-01-01 00:00:01,367409990,0,231,511,126
2009-01-01 00:00:01,538003423,11,57,113,128
2009-01-01 00:00:01,664445000,16,119,511,127
2009-01-01 00:00:02,367409990,0,249,511,128
2009-01-01 00:00:02,538003423,12,59,113,127
2009-01-01 00:00:02,664445000,18,211,334,128
```

```
2009-01-01 00:00:03,367409990,2,247,128,127
2009-01-01 00:00:03,538003423,12,69,511,128
2009-01-01 00:00:03,664445000,19,147,365,127
2009-01-01 00:00:05,367409990,5,254,119,126
2009-01-01 00:00:05,538003423,13,79,118,128
2009-01-01 00:00:05,664445000,20,163,511,127
2009-01-01 00:00:07,367409990,6,245,110,126
2009-01-01 00:00:07,538003423,18,99,128,126
2009-01-01 00:00:07,664445000,21,162,375,128
```

```
Owner: rsn1ID
Vessel: 367409990
Status: Undef
TimeSeries:
        Timestamp           | SOG | COG | Heading | ROT |
Thu Jan 01 00:00:01 PST 2009  | (rsn1ID, obs1ID, snr2ID, 0)  | (rsn1ID, obs1ID, snr1ID, 245)  | (rsn1ID, obs1ID, snr2ID, 511)  | (rsn1ID, obs1ID, snr2ID, 128)  |
Thu Jan 01 00:00:02 PST 2009  | (rsn1ID, obs1ID, snr2ID, 0)  | (rsn1ID, obs1ID, snr1ID, 228)  | (rsn1ID, obs1ID, snr1ID, 511)  | (rsn1ID, obs1ID, snr2ID, 128)  |
Thu Jan 01 00:00:03 PST 2009  | (rsn1ID, obs2ID, snr3ID, 0)  | (rsn1ID, obs2ID, snr3ID, 240)  | (rsn1ID, obs2ID, snr4ID, 511)  | (rsn1ID, obs2ID, snr4ID, 128)  |
Thu Jan 01 00:00:04 PST 2009  | (rsn1ID, obs1ID, snr2ID, 2)  | (rsn1ID, obs1ID, snr1ID, 248)  | (rsn1ID, obs1ID, snr2ID, 129)  | (rsn1ID, obs1ID, snr2ID, 128)  |
Thu Jan 01 00:00:05 PST 2009  | (rsn1ID, obs2ID, snr3ID, 3)  | (rsn1ID, obs2ID, snr4ID, 238)  | (rsn1ID, obs1ID, snr2ID, 511)  | (rsn1ID, obs2ID, snr3ID, 128)  |
Thu Jan 01 00:00:06 PST 2009  | (rsn1ID, obs2ID, snr3ID, 3)  | (rsn1ID, obs2ID, snr3ID, 225)  | (rsn1ID, obs2ID, snr4ID, 152)  | (rsn1ID, obs2ID, snr3ID, 121)  |
Thu Jan 01 00:00:07 PST 2009  | (rsn1ID, obs2ID, snr3ID, 4)  | (rsn1ID, obs2ID, snr4ID, 290)  | (rsn1ID, obs2ID, snr4ID, 511)  | (rsn1ID, obs2ID, snr3ID, 126)  |
Thu Jan 01 00:00:08 PST 2009  | (rsn1ID, obs2ID, snr3ID, 4)  | (rsn1ID, obs2ID, snr4ID, 295)  | (rsn1ID, obs2ID, snr4ID, 167)  | (rsn1ID, obs2ID, snr3ID, 128)  |
Thu Jan 01 00:00:09 PST 2009  | (rsn1ID, obs1ID, snr2ID, 4)  | (rsn1ID, obs1ID, snr1ID, 250)  | (rsn1ID, obs1ID, snr1ID, 93)  | (rsn1ID, obs1ID, snr2ID, 128)  |
Thu Jan 01 00:00:10 PST 2009  | (rsn1ID, obs1ID, snr2ID, 4)  | (rsn1ID, obs1ID, snr1ID, 261)  | (rsn1ID, obs1ID, snr2ID, 142)  | (rsn1ID, obs1ID, snr2ID, 128)  |


Owner: rsn1ID
Vessel: 664445000
Status: Undef
TimeSeries:
        Timestamp           | SOG | COG | Heading | ROT |
Thu Jan 01 00:00:06 PST 2009  | (rsn1ID, obs2ID, snr3ID, 19)  | (rsn1ID, obs2ID, snr4ID, 157)  | (rsn1ID, obs2ID, snr4ID, 324)  | (rsn1ID, obs2ID, snr4ID, 128)  |
Thu Jan 01 00:00:07 PST 2009  | (rsn1ID, obs1ID, snr2ID, 16)  | (rsn1ID, obs2ID, snr3ID, 182)  | (rsn1ID, obs2ID, snr4ID, 335)  | (rsn1ID, obs2ID, snr3ID, 128)  |
Thu Jan 01 00:00:08 PST 2009  | (rsn1ID, obs2ID, snr3ID, 21)  | (rsn1ID, obs2ID, snr3ID, 175)  | (rsn1ID, obs2ID, snr4ID, 356)  | (rsn1ID, obs2ID, snr3ID, 126)  |
Thu Jan 01 00:00:09 PST 2009  | (rsn1ID, obs2ID, snr3ID, 23)  | (rsn1ID, obs2ID, snr3ID, 169)  | (rsn1ID, obs2ID, snr4ID, null)  | (rsn1ID, obs2ID, snr3ID, 127)  |
Thu Jan 01 00:00:10 PST 2009  | (rsn1ID, obs1ID, snr2ID, 18)  | (rsn1ID, obs1ID, snr1ID, 172)  | (rsn1ID, obs1ID, snr1ID, null)  | (rsn1ID, obs1ID, snr2ID, 121)  |


Owner: rsn1ID
Vessel: 538003423
Status: Undef
TimeSeries:
        Timestamp           | SOG | COG | Heading | ROT |
Thu Jan 01 00:00:01 PST 2009  | (rsn1ID, obs2ID, snr3ID, 11)  | (rsn1ID, obs2ID, snr4ID, 67)  | (rsn1ID, obs2ID, snr4ID, 12)  | (rsn1ID, obs2ID, snr3ID, 127)  |
Thu Jan 01 00:00:02 PST 2009  | (rsn1ID, obs1ID, snr2ID, 12)  | (rsn1ID, obs1ID, snr1ID, 68)  | (rsn1ID, obs1ID, snr2ID, 12)  | (rsn1ID, obs1ID, snr2ID, 126)  |
Thu Jan 01 00:00:03 PST 2009  | (rsn1ID, obs2ID, snr3ID, 14)  | (rsn1ID, obs2ID, snr3ID, 65)  | (rsn1ID, obs2ID, snr4ID, 19)  | (rsn1ID, obs2ID, snr3ID, 110)  |
Thu Jan 01 00:00:04 PST 2009  | (rsn1ID, obs1ID, snr2ID, 15)  | (rsn1ID, obs1ID, snr1ID, 74)  | (rsn1ID, obs1ID, snr2ID, 511)  | (rsn1ID, obs1ID, snr2ID, 128)  |
Thu Jan 01 00:00:05 PST 2009  | (rsn1ID, obs2ID, snr3ID, 15)  | (rsn1ID, obs2ID, snr3ID, 71)  | (rsn1ID, obs2ID, snr4ID, 511)  | (rsn1ID, obs2ID, snr3ID, 121)  |
Thu Jan 01 00:00:06 PST 2009  | (rsn1ID, obs1ID, snr2ID, 17)  | (rsn1ID, obs1ID, snr1ID, 69)  | (rsn1ID, obs1ID, snr1ID, 12)  | (rsn1ID, obs1ID, snr2ID, 128)  |
Thu Jan 01 00:00:07 PST 2009  | (rsn1ID, obs2ID, snr3ID, 17)  | (rsn1ID, obs2ID, snr3ID, 84)  | (rsn1ID, obs2ID, snr4ID, 511)  | (rsn1ID, obs2ID, snr3ID, 127)  |
Thu Jan 01 00:00:08 PST 2009  | (rsn1ID, obs1ID, snr2ID, 19)  | (rsn1ID, obs1ID, snr1ID, 84)  | (rsn1ID, obs1ID, snr2ID, 14)  | (rsn1ID, obs1ID, snr2ID, 124)  |
Thu Jan 01 00:00:10 PST 2009  | (rsn1ID, obs2ID, snr3ID, 19)  | (rsn1ID, obs2ID, snr3ID, 89)  | (rsn1ID, obs2ID, snr4ID, 21)  | (rsn1ID, obs2ID, snr3ID, 120)  |
```

```
Owner: rsn2ID
Vessel: 367409990
Status: Undef
TimeSeries:
        Timestamp           | SOG | COG | Heading | ROT |
Thu Jan 01 00:00:01 PST 2009  | (rsn2ID, obs4ID, snr7ID, 0)  | (rsn2ID, obs4ID, snr7ID, 231)  | (rsn2ID, obs4ID, snr7ID, 511)  | (rsn2ID, obs4ID, snr7ID, 126)  |
Thu Jan 01 00:00:02 PST 2009  | (rsn2ID, obs4ID, snr7ID, 0)  | (rsn2ID, obs3ID, snr6ID, 212)  | (rsn2ID, obs4ID, snr7ID, 511)  | (rsn2ID, obs4ID, snr7ID, 128)  |
Thu Jan 01 00:00:03 PST 2009  | (rsn2ID, obs4ID, snr7ID, 2)  | (rsn2ID, obs4ID, snr7ID, 247)  | (rsn2ID, obs4ID, snr7ID, 128)  | (rsn2ID, obs4ID, snr7ID, 127)  |
Thu Jan 01 00:00:04 PST 2009  | (rsn2ID, obs3ID, snr5ID, 2)  | (rsn2ID, obs3ID, snr6ID, 234)  | (rsn2ID, obs3ID, snr5ID, 132)  | (rsn2ID, obs3ID, snr6ID, 127)  |
Thu Jan 01 00:00:05 PST 2009  | (rsn2ID, obs4ID, snr7ID, 5)  | (rsn2ID, obs4ID, snr7ID, 254)  | (rsn2ID, obs4ID, snr7ID, 119)  | (rsn2ID, obs4ID, snr7ID, 126)  |
Thu Jan 01 00:00:06 PST 2009  | (rsn2ID, obs3ID, snr5ID, 4)  | (rsn2ID, obs3ID, snr6ID, 225)  | (rsn2ID, obs3ID, snr5ID, 511)  | (rsn2ID, obs3ID, snr6ID, 127)  |
Thu Jan 01 00:00:07 PST 2009  | (rsn2ID, obs4ID, snr7ID, 6)  | (rsn2ID, obs4ID, snr7ID, 245)  | (rsn2ID, obs4ID, snr7ID, 110)  | (rsn2ID, obs4ID, snr7ID, 126)  |
Thu Jan 01 00:00:08 PST 2009  | (rsn2ID, obs3ID, snr5ID, 5)  | (rsn2ID, obs3ID, snr6ID, 254)  | (rsn2ID, obs3ID, snr5ID, 511)  | (rsn2ID, obs3ID, snr6ID, 123)  |
Thu Jan 01 00:00:09 PST 2009  | (rsn2ID, obs3ID, snr5ID, 5)  | (rsn2ID, obs3ID, snr5ID, 287)  | (rsn2ID, obs3ID, snr5ID, 116)  | (rsn2ID, obs3ID, snr6ID, 123)  |


Owner: rsn2ID
Vessel: 664445000
Status: Undef
TimeSeries:
        Timestamp           | SOG | COG | Heading | ROT |
Thu Jan 01 00:00:01 PST 2009  | (rsn2ID, obs4ID, snr7ID, 16)  | (rsn2ID, obs4ID, snr7ID, 119)  | (rsn2ID, obs4ID, snr7ID, 511)  | (rsn2ID, obs4ID, snr7ID, 127)  |
Thu Jan 01 00:00:02 PST 2009  | (rsn2ID, obs4ID, snr7ID, 18)  | (rsn2ID, obs3ID, snr5ID, 119)  | (rsn2ID, obs4ID, snr7ID, 334)  | (rsn2ID, obs4ID, snr7ID, 128)  |
Thu Jan 01 00:00:03 PST 2009  | (rsn2ID, obs4ID, snr7ID, 19)  | (rsn2ID, obs4ID, snr7ID, 147)  | (rsn2ID, obs4ID, snr7ID, null)  | (rsn2ID, obs4ID, snr7ID, 127)  |
Thu Jan 01 00:00:04 PST 2009  | (rsn2ID, obs3ID, snr5ID, 15)  | (rsn2ID, obs3ID, snr5ID, 124)  | (rsn2ID, obs3ID, snr5ID, 334)  | (rsn2ID, obs3ID, snr6ID, 128)  |
Thu Jan 01 00:00:05 PST 2009  | (rsn2ID, obs4ID, snr7ID, 20)  | (rsn2ID, obs4ID, snr7ID, 163)  | (rsn2ID, obs4ID, snr7ID, 511)  | (rsn2ID, obs4ID, snr7ID, 127)  |
Thu Jan 01 00:00:06 PST 2009  | (rsn2ID, obs3ID, snr5ID, 17)  | (rsn2ID, obs3ID, snr5ID, 137)  | (rsn2ID, obs3ID, snr5ID, null)  | (rsn2ID, obs3ID, snr6ID, 128)  |
Thu Jan 01 00:00:07 PST 2009  | (rsn2ID, obs4ID, snr7ID, 21)  | (rsn2ID, obs4ID, snr7ID, 162)  | (rsn2ID, obs4ID, snr7ID, null)  | (rsn2ID, obs4ID, snr7ID, 128)  |
Thu Jan 01 00:00:08 PST 2009  | (rsn2ID, obs3ID, snr5ID, 21)  | (rsn2ID, obs3ID, snr6ID, 139)  | (rsn2ID, obs3ID, snr5ID, 511)  | (rsn2ID, obs3ID, snr6ID, 128)  |
Thu Jan 01 00:00:09 PST 2009  | (rsn2ID, obs3ID, snr5ID, 23)  | (rsn2ID, obs3ID, snr5ID, 153)  | (rsn2ID, obs3ID, snr5ID, 511)  | (rsn2ID, obs3ID, snr6ID, 125)  |


Owner: rsn2ID
Vessel: 538003423
Status: Undef
TimeSeries:
        Timestamp           | SOG | COG | Heading | ROT |
Thu Jan 01 00:00:01 PST 2009  | (rsn2ID, obs4ID, snr7ID, 11)  | (rsn2ID, obs4ID, snr7ID, 57)  | (rsn2ID, obs4ID, snr7ID, 113)  | (rsn2ID, obs4ID, snr7ID, 128)  |
Thu Jan 01 00:00:02 PST 2009  | (rsn2ID, obs4ID, snr7ID, 12)  | (rsn2ID, obs4ID, snr7ID, 59)  | (rsn2ID, obs4ID, snr7ID, 113)  | (rsn2ID, obs4ID, snr7ID, 127)  |
Thu Jan 01 00:00:03 PST 2009  | (rsn2ID, obs4ID, snr7ID, 12)  | (rsn2ID, obs4ID, snr7ID, 69)  | (rsn2ID, obs4ID, snr7ID, 511)  | (rsn2ID, obs4ID, snr7ID, 128)  |
Thu Jan 01 00:00:04 PST 2009  | (rsn2ID, obs3ID, snr5ID, 15)  | (rsn2ID, obs3ID, snr6ID, 77)  | (rsn2ID, obs3ID, snr5ID, 123)  | (rsn2ID, obs3ID, snr6ID, 128)  |
Thu Jan 01 00:00:05 PST 2009  | (rsn2ID, obs4ID, snr7ID, 13)  | (rsn2ID, obs4ID, snr7ID, 79)  | (rsn2ID, obs4ID, snr7ID, 118)  | (rsn2ID, obs4ID, snr7ID, 128)  |
Thu Jan 01 00:00:06 PST 2009  | (rsn2ID, obs3ID, snr5ID, 17)  | (rsn2ID, obs3ID, snr5ID, 65)  | (rsn2ID, obs3ID, snr5ID, 156)  | (rsn2ID, obs3ID, snr6ID, 128)  |
Thu Jan 01 00:00:07 PST 2009  | (rsn2ID, obs4ID, snr7ID, 18)  | (rsn2ID, obs4ID, snr7ID, 99)  | (rsn2ID, obs4ID, snr7ID, 128)  | (rsn2ID, obs4ID, snr7ID, 126)  |
Thu Jan 01 00:00:08 PST 2009  | (rsn2ID, obs3ID, snr5ID, 17)  | (rsn2ID, obs3ID, snr6ID, 79)  | (rsn2ID, obs3ID, snr5ID, 114)  | (rsn2ID, obs3ID, snr6ID, 128)  |
Thu Jan 01 00:00:09 PST 2009  | (rsn2ID, obs3ID, snr5ID, 18)  | (rsn2ID, obs3ID, snr5ID, 91)  | (rsn2ID, obs3ID, snr5ID, 110)  | (rsn2ID, obs3ID, snr6ID, 128)  |
```

Chapter 7

Conclusions and Future Work

Providing safety and security for the maritime domain against potential threats and illegal activities needs constant monitoring and analyzing. Examining different marine traffic monitoring systems and going in depth with how they operate to observe and interpret real-world situations are essential to have a deeper understanding of their capabilities and limitations. Fusing marine traffic data from various sources provides a more, *i)* coherent and comprehensive, and *ii)* accurate and reliable view of the real-world pertaining to the maritime domain.

Intelligent systems are crucial for interactive situation analysis and dynamic decision-making, responding to complex real-world situations calls for innovative methodical and formal approaches to economically develop robust and scalable solutions. In this thesis, we propose a formal semantic framework for maritime situation analysis based on the work presented in [32, 37]. For this purpose, we use the Abstract State Machine method to; *i)* seamlessly bridge the gap between abstract requirements and formal specifications, and *ii)* formally express the underlying design concepts, assumptions, and constraints. At first, these requirements and assumptions are interpreted as the core of the situation analysis ground model; which is then refined vertically and horizontally to a more concrete model.

Maritime surveillance data is interpreted as sequences of discrete observations over time that render vessel trajectories. Consecutive data points referring to the same object form a time series used for analyzing and reasoning about vessel behavior. However, more than one time series can initially belong to one object (i.e., vessel). In order to describe how a number of time series related to one object are fused into a single comprehensive multivariate time series, we formally defined a time series *structure* along with a set *operations* (e.g., horizontal merge or vertical merge) to render the *trajectory* of vessels. As proof of concept, the core components of the framework are implemented in CoreASM and Java to demonstrate the process of generating vessel trajectories from diverse sources of real-world data. In our implementation, CoreASM controls the information fusion and situation analysis process while Java deals with data and all computational components. We also extract some test

data from real-world AIS datasets collected by U.S. Coast Guard Services (available at www.marinecadastre.gov) to experimentally validate the framework and the consistency between the high-level requirements, the abstract ground model and the executable model.

7.1 Future Work

- **Vertical Refinement:** Enriching each component by capturing more concrete domain-specific requirements to address real-world constraints and demands. For instance, proposing an algorithm which find the optimal "close" timestamps.
- **Conflict Resolution and Confidence Values:** Proposing a more concrete model for dealing with real-world uncertainties (e.g., sensor error, missing data, etc.) as well as assigning confidence values to them.
- **Distributed Model:** Expanding the framework to support more than one reasoner in a distributed manner. Distributed agreement protocols should be proposed to provide a global common view among all reasoners.

Expected outcomes of this thesis are the design of a generic system and service model for situation analysis in the maritime domain as well as a working prototype to be tested in a simulated operational environment.

Bibliography

[1] All About AIS Website. Available electronically at `http://www.allaboutais.com` (Last visited in April 2015).

[2] Iman Attarzadeh and Siew Hock Ow. Project management practices: the criteria for success or failure. *Communications of the IBIMA*, 1:234–241, 2008.

[3] Automatic Identification System *Wikipedia, the free encyclopedia.* Available electronically at `https://en.wikipedia.org/wiki/Automatic_Identification_System` (Last visited in March 2015).

[4] Franz Baader, Andreas Bauer, Peter Baumgartner, Anne Cregan, Alfredo Gabaldon, Krystian Ji, Kevin Lee, David Rajaratnam, and Rolf Schwitter. A novel architecture for situation awareness systems. In *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 77–92. Springer, 2009.

[5] BBC News Website. Available electronically at `http://www.bbc.com/news/technology-24586394` (Last visited in August 2014).

[6] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic (TOCL)*, 4(4):578–651, 2003.

[7] Egon Börger, Uwe Glässer, and Wolfgang Muller. A formal definition of an abstract vhdlí93 simulator by ea-machines. In *Formal Semantics for VHDL*, pages 107–139. Springer, 1995.

[8] Egon Börger and Robert F. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[9] Éloi Bossé, Jean Roy, and Steve Wark. Concepts, models, and tools for information fusion, artech house. *Inc.*, 27:28–29, 2007.

[10] Chris Chatfield. *The Analysis of Time Series: An Introduction.* Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis, sixth edition edition, 1996.

[11] Mariusz Chmielewski. Ontology applications for achieving situation awareness in military decision support systems. In *Computational Collective Intelligence. Semantic Web, Social Networks and Multiagent Systems*, pages 528–539. Springer, 2009.

[12] James Douglas Clines. Method and system for marine vessel tracking system, December 2 2003. US Patent 6,658,349.

[13] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z*, volume 30. Springer, 2001.

[14] Mica R. Endsley and Daniel J. Garland. *Situation awareness analysis and measurement*. CRC Press, 2000.

[15] Roozbeh Farahbod. *CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems*. PhD thesis, Simon Fraser University, 2009.

[16] Roozbeh Farahbod, Vladimir Avram, Uwe Glässer, and Adel Guitouni. A formal engineering approach to high-level design of situation analysis decision support systems. In *Formal Methods and Software Engineering*, pages 211–226. Springer, 2011.

[17] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. Coreasm: An extensible asm execution engine. *Fundamenta Informaticae*, 77(1-2):71–104, 2007.

[18] Roozbeh Farahbod, Vincenzo Gervasi, Uwe Glässer, and George Ma. Coreasm plug-in architecture. In *Rigorous Methods for Software Construction and Analysis*, pages 147–169. Springer, 2009.

[19] Roozbeh Farahbod and Uwe Glässer. The coreasm modeling framework. *Software: Practice and Experience*, 41(2):167–178, 2011.

[20] Roozbeh Farahbod, Uwe Glässer, Eloi Bossé, and Adel Guitouni. Integrating abstract state machines and interpreted systems for situation analysis decision support design. In *Information Fusion, 2008 11th International Conference on*, pages 1–8. IEEE, 2008.

[21] Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. An abstract machine architecture for web service based business process management. *International Journal of Business Process Integration and Management*, 1(4):279–291, 2006.

[22] Vincenzo Gervasi and Roozbeh Farahbod. Jasmine: Accessing java code from coreasm. In *Rigorous Methods for Software Construction and Analysis*, pages 170–186. Springer, 2009.

[23] Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. The formal semantics of sdl-2000: Status and perspectives. *Comput. Netw.*, 42(3):343–358, June 2003.

[24] Uwe Glässer, Yuri Gurevich, and Margus Veanes. Abstract communication model for distributed systems. *Software Engineering, IEEE Transactions on*, 30(7):458–472, 2004.

[25] Uwe Glässer, Piper Jackson, Ali Khalili Araghi, and Hamed Yaghoubi Shahir. Intelligent decision support for marine safety and security operations. In *Intelligence and Security Informatics (ISI), 2010 IEEE International Conference on*, pages 101–107. IEEE, 2010.

[26] Uwe Glässer and Hamed Yaghoubi Shahir. *ASM/CoreASM Tutorial*. Software Technology Lab, School of Computing Science, Simon Fraser University, 2013.

[27] Yuri Gurevich and James K Huggins. The railroad crossing problem: an experiment with instantaneous actions and immediate reactions. In *Computer Science Logic*, pages 266–290. Springer, 1996.

[28] International Maritime Organization Website. Available electronically at `http://www.imo.org/blast/blastDataHelper.asp?data_id=29093&filename=1367.pdf` (Last visited in April 2015).

[29] Gabriel Jakobson, Lundy Lewis, John Buford, and Col Ed Sherman. Battlespace situation analysis: the dynamic cbr approach. In *Military Communications Conference, 2004. MILCOM 2004. 2004 IEEE*, volume 2, pages 941–947. IEEE, 2004.

[30] Anne-Laure Jousselme and Patrick Maupin. Interpreted systems for situation analysis. In *Information Fusion, 2007 10th International Conference on*, pages 1–11. IEEE, 2007.

[31] Robb Klashner and Sameh Sabet. A dss design model for complex problems: Lessons from mission critical infrastructure. *Decision Support Systems*, 43(3):990–1013, 2007.

[32] Dale A Lambert. A blueprint for higher-level fusion systems. *Information Fusion*, 10(1):6–24, 2009.

[33] Marine radar *Wikipedia, the free encyclopedia.* Available electronically at `https://en.wikipedia.org/wiki/Marine_radar` (Last visited in May 2015).

[34] MarineTraffic Website. Available electronically at `http://www.marinetraffic.com/en/p/faq` (Last visited in April 2015).

[35] Maritime and Port Authority of Singapore Website. Available electronically at `http://www.mpa.gov.sg/web/wcm/connect/www/7b6c139b-f004-4812-acf2-ac3738230067/pc05-21.pdf?MOD=AJPERES` (Last visited in April 2015).

[36] J McDermid. Science of software design: Architectures for evolvable, dependable systems. In *NSF Workshop on the Science of Design: Software and Software-Intensive Systems, Airlie Center, VA*, 2003.

[37] Narek Nalbandyan, Uwe Glässer, Hamed Yaghoubi Shahir, and Hans Wehn. Distributed situation analysis - a formal semantic framework. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, pages 158–173, 2014.

[38] National Geospatial-Intelligence Agency Website. Available electronically at `http://msi.nga.mil/MSISiteContent/StaticFiles/NAV_PUBS/RNM/310ch1.pdf` (Last visited in January 2016).

[39] ORBCOMM Inc. Website. Available electronically at `http://www.orbcomm.com/networks/satellite-ais` (Last visited in April 2015).

[40] Elvinia Riccobene and Joachim Schmid. Capturing requirements by abstract state machines: The light control case study. *Journal of Universal Computer Science*, 6(7):597–620, 2000.

[41] River Information Services Website. Available electronically at http://ris.vlaanderen.be/html_en/AIS/ais_vragen.html#klasseAenB (Last visited in March 2015).

[42] Hamed Yaghoubi Shahir, Uwe Glässer, Narek Nalbandyan, and Hans Wehn. Maritime situation analysis. In *2013 IEEE International Conference on Intelligence and Security Informatics*, 2013.

[43] Hamed Yaghoubi Shahir, Uwe Glässer, Amir Yaghoubi Shahir, and Hans Wehn. Maritime situation analysis framework: Vessel interaction classification and anomaly detection. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1279–1289. IEEE, 2015.

[44] Ship AIS Website. Available electronically at http://www.shipais.com/doc/Pifaq/1/22/ (Last visited in March 2015).

[45] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java virtual machine: definition, verification, validation.* Springer Science & Business Media, 2012.

[46] Alan N. Steinberg, Christopher L. Bowman, and Franklin E. White. Revisions to the jdl data fusion model. In *AeroSense'99*, pages 430–441. International Society for Optics and Photonics, 1999.

[47] Commander Brian J. Tetreault. Use of the automatic identification system (ais) for maritime domain awareness (mda). In *OCEANS, 2005. Proceedings of MTS/IEEE*, pages 1590–1594. IEEE, 2005.

[48] The Bosun's Mate Website. Available electronically at http://www.bosunsmate.org/ais/ (Last visited in March 2015).

[49] Time Series *Wikipedia, the free encyclopedia.* Available electronically at http://en.wikipedia.org/wiki/time_series (Last visited in August 2014).

[50] Trend Micro Website. Available electronically at http://blog.trendmicro.com/trendlabs-security-intelligence/captain-where-is-your-ship-compromising-vessel-tracking-systems/.

[51] U.S. Coast Guard Navigation Center. Available electronically at http://www.navcen.uscg.gov (Last visited in March 2015).

[52] Vessel Traffic Service *Wikipedia, the free encyclopedia.* Available electronically at https://en.wikipedia.org/wiki/Vessel_traffic_service (Last visited in April 2015).

[53] Franklin E. White Jr. Data fusion lexicon, joint directors of laboratories. *Technical panel for C*, 3, 1987.

Appendix A

Implementation Details

A.1 Time Series Operations

A.1.1 Horizontal Merge (Initial Definition)

```
public void horizontalMergeWIT(TimeSeries ts) throws IOException {
   if (ts.getHead().size() != ts.getBody().get(0).getData().size())
      throw new IOException("Horizontal Merge Error: Head size and Data size of the merging
          timeseries, " + ts + ", are NOT matching.");

      if (this.head.isEmpty() && this.body.isEmpty()) {
         head.addAll(ts.getHead());
         body.addAll(ts.getBody());
      }

      else if (this.getTimestampSet().equals(ts.getTimestampSet())) {
         ArrayList<Feature> tempHead = new ArrayList<Feature>();
         tempHead.addAll(head);

         int headi = 0;
         int tsHeadi = 0;
         int featurei = 0;
         int tempHeadi = 0;

         while ((tempHeadi < tempHead.size()) && (tsHeadi < ts.getHead().size())) {
            if (tempHead.get(tempHeadi).equals(ts.getHead().get(tsHeadi))) {
               tempHeadi++;
               tsHeadi++;
            }
            else {
               if (tempHead.get(tempHeadi).equals(Globals.FEATURES.get(featurei))) {
                  tempHeadi++;
                  featurei++;
               }
               else if (ts.getHead().get(tsHeadi).equals(Globals.FEATURES.get(featurei))) {
                  tempHead.add(tempHeadi, Globals.FEATURES.get(featurei));
                  tempHeadi++;
                  tsHeadi++;
                  featurei++;
```

```
            }
            else {
               featurei++;
            }
         }
      }
      if (tempHeadi >= tempHead.size()) {
         while (tsHeadi < ts.getHead().size()) {
            tempHead.add(ts.getHead().get(tsHeadi));

            tsHeadi++;
         }
      }

      ArrayList<Tuple> tempBody = new ArrayList<Tuple>();

      for (int bodyi = 0; bodyi < body.size(); bodyi++) {
         Tuple tuple = new Tuple(tempHead.size());
         tuple.setTimestamp(body.get(bodyi).getTimestamp());

         tempHeadi = 0;
         headi = 0;
         tsHeadi = 0;

         for (tempHeadi = 0; tempHeadi < tempHead.size(); tempHeadi++) {
            if ((head.get(headi).equals(tempHead.get(tempHeadi))) &&
                 (ts.getHead().get(tsHeadi)
            .equals(tempHead.get(tempHeadi)))) {
               tuple.addRecords2CellOfData(body.get(bodyi).getData().get(headi),
                    tempHeadi);
               tuple.addRecords2CellOfData(ts.getBody().get(bodyi).getData().get(tsHeadi),
                    tempHeadi);

               if (headi < head.size()-1) headi++;
               if (tsHeadi < ts.getHead().size()-1) tsHeadi++;
            }
            else if ((head.get(headi).equals(tempHead.get(tempHeadi))) &&
                 !(ts.getHead().get(tsHeadi)
            .equals(tempHead.get(tempHeadi)))) {
               tuple.addRecords2CellOfData(body.get(bodyi).getData().get(headi),
                    tempHeadi);

               if (headi < head.size()-1) headi++;
            }
            else if (!(head.get(headi).equals(tempHead.get(tempHeadi))) &&
                 (ts.getHead().get(tsHeadi)
            .equals(tempHead.get(tempHeadi)))) {
               tuple.addRecords2CellOfData(ts.getBody().get(bodyi).getData().get(tsHeadi),
                    tempHeadi);

               if (tsHeadi < ts.getHead().size()-1) tsHeadi++;
            }
         }
         tempBody.add(tuple);
      }
      this.head = tempHead;
      this.body = tempBody;
   }
```

```
    else
      throw new IOException("Horizontal Merge Error: The timestamps of two time series
          are NOT identical.");

}
```

A.1.2 Vertical Merge (Initial Definition)

```
public void verticalMergeWIH(TimeSeries ts) throws IOException {
  if (ts.getHead().size() != ts.getBody().get(0).getData().size())
    throw new IOException("Vertical Merge Error: Head size and Data size of the merging
        time series, " + ts + ", are NOT matching.");

    if (this.head.isEmpty() && this.body.isEmpty()) {
      head.addAll(ts.getHead());
      body.addAll(ts.getBody());
    }

    else if (this.head.equals(ts.getHead())) {
      ArrayList<Tuple> tempBody = new ArrayList<Tuple>();

      int bodyi = 0;
      int tsBodyi = 0;
      int celli;

      while ((bodyi < body.size()) && (tsBodyi < ts.getBody().size())) {
        if
            (body.get(bodyi).getTimestamp().before(ts.getBody().get(tsBodyi).getTimestamp()))
            {
          tempBody.add(body.get(bodyi));

          bodyi++;
        }
        else if
            (body.get(bodyi).getTimestamp().after(ts.getBody().get(tsBodyi).getTimestamp()))
            {
          tempBody.add(ts.getBody().get(tsBodyi));

          tsBodyi++;
        }
        else {
          tempBody.add(body.get(bodyi));

          bodyi++;

          celli = 0;
          while (celli < ts.getBody().get(tsBodyi).getData().size()) {
            tempBody.get(tempBody.size()-1)
            .addRecords2CellOfData(ts.getBody().get(tsBodyi).getRecordsOfCell(celli),
                celli);

            celli++;
          }
          tsBodyi++;
        }
```

```
      }
      if (bodyi >= body.size()) {
         while (tsBodyi < ts.getBody().size()) {
            tempBody.add(ts.getBody().get(tsBodyi));
            tsBodyi++;
         }
      }
      else if (tsBodyi >= ts.getBody().size()) {
         while (bodyi < body.size()) {
            tempBody.add(body.get(bodyi));
            bodyi++;
         }
      }

      this.body = new ArrayList<Tuple>(tempBody);
   }
   else
      throw new IOException("Vertical Merge Error: The head of two time series are NOT
          identical.");

}
```

A.2 Initialization of Data Structures

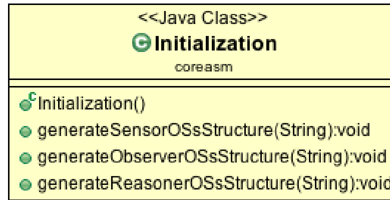


Figure A.1: Initialization Class Diagram

```
public class Initialization {

   public void generateSensorOSsStructure(String snr) {
      Globals.rawOSs.put(snr, new HashMap<Long, ArrayList<ObservationSegment>>());
   }

   public void generateObserverOSsStructure(String obs) {
      Globals.cleanedOSs.put(obs, new HashMap<Long, ArrayList<ObservationSegment>>());

      Globals.mergedOSs.put(obs, new HashMap<Long, ArrayList<ObservationSegment>>());

   }

   public void generateReasonerOSsStructure(String rsn) {
      Globals.unifiedOSs.put(rsn, new HashMap<String, HashMap<Long,
          ArrayList<ObservationSegment>>>());

      Globals.fusedOSs.put(rsn, new HashMap<Long, ObservationSegment>());
      Globals.endingTimestamp.put(rsn, new HashMap<Long, HashMap<String, Date>>());

      Globals.cfusedOSs.put(rsn, new HashMap<Long, ObservationSegment>());
   }

}
```

A.3 Input/Output Manager

Figure A.2: I/O Manager Class Diagram

```
public class IOManager {
   //READ FROM A COMMA-SEPARATED VALUES FILE:
   public void readFromCSVFile(String sensorID, int t) throws ParseException,
       NumberFormatException, IOException {

      BufferedReader bufferReader = null;
      String strLine = null;

      Calendar time = Calendar.getInstance();
      time.setTime(Globals.BASE_TIME.getTime());
      time.add(Calendar.SECOND, t);

      String file = Globals.SENSOR_FILENAME_PREFIX + sensorID.trim() +
          Globals.SENSOR_FILENAME_POSTFIX;

      FileReader inputFile = new FileReader(file);
      bufferReader = new BufferedReader(inputFile);

      ArrayList<Feature> head = new ArrayList<Feature>();
      strLine = bufferReader.readLine();
      for (String feature : strLine.split(",")) {
         if (feature.equals("Timestamp")) /*Nothing*/;
         else if (feature.equals("MMSI")) /*Nothing*/;
         else if (feature.equals("SOG")) head.add(Feature.SOG);
         else if (feature.equals("COG")) head.add(Feature.COG);
         else if (feature.equals("Heading")) head.add(Feature.Heading);
         else if (feature.equals("ROT")) head.add(Feature.ROT);
         else head.add(Feature.UNDEFINED);
      }

      while ((strLine = bufferReader.readLine()) != null) {
         String[] data = strLine.split(",");

         String timeStamp = data[0]; //READING TIMESTAMP

         if (Globals.DATE_FORMAT.parse(timeStamp).equals(time.getTime())) {
            ObservationSegment os = new ObservationSegment();
            os.setOwner(sensorID);
            os.setStatus(Status.Undef);

            TimeSeries timeSeries = new TimeSeries(head);
            Tuple tuple = new Tuple(head.size());

            tuple.setTimestamp(Globals.DATE_FORMAT.parse(timeStamp));
```

```
        String objectID = data[1]; //READING OBJECT ID
        Vessel vessel = new Vessel();
        if (!(objectID.isEmpty())) {
          vessel.setID(Long.parseLong(objectID));
          os.setObject(vessel);
        }
        else
          os.setObject(null);

        for (int celli = 2; celli < data.length; celli++) {
          String element = data[celli];

          Record record = new Record();
          record.setSensorID(sensorID);

          if (Utilities.isNumeric(element))
            record.setValue(Integer.parseInt(element));
          else
            record.setValue(null);

          tuple.addRecord2CellOfData(record, celli-2);

        }
        timeSeries.addToBody(tuple);
        os.setTimeSeries(timeSeries);

        if (Globals.rawOSs.containsKey(sensorID)){
          if (Globals.rawOSs.get(sensorID).containsKey(os.getObject().getID()))
            Globals.rawOSs.get(sensorID).get(os.getObject().getID()).add(os);
          else {
            ArrayList<ObservationSegment> tempList = new
                ArrayList<ObservationSegment>();
            tempList.add(os);

            Globals.rawOSs.get(sensorID).put(os.getObject().getID(), tempList);
          }
        }
        else {
           throw new IOException("I/O Manager Error: The Sensor ID, " + sensorID + ",
               is not valid.");
        }
      }
    }
  }

}
```

A.4 Implementation of Rules in Java

A.4.1 CleanObservations

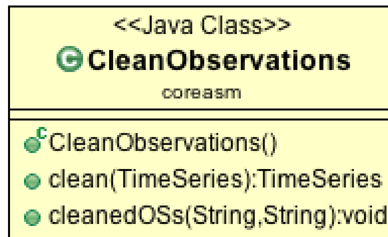


Figure A.3: Clean Observations Class Diagram

```
public class CleanObservations {

   public void cleanedOSs(String observerID, String sensorID) throws IOException {
      if (!Globals.rawOSs.containsKey(sensorID))
         throw new IOException("Clean Observations Error: The sensor ID, " + sensorID + ",
             is NOT valid.");
      if (!Globals.cleanedOSs.containsKey(observerID))
         throw new IOException("Clean Observations Error: The observer ID, " + observerID +
             ", is NOT valid.");

      if (!(Globals.rawOSs.get(sensorID).isEmpty())) {
         for (Entry<Long, ArrayList<ObservationSegment>> entry :
             Globals.rawOSs.get(sensorID).entrySet()) {
            for (ObservationSegment ros : entry.getValue()) {
               if (!(ros.getStatus().equals(Status.Cleaned))) {
                  ObservationSegment cos = new ObservationSegment();

                  cos.setOwner(observerID);
                  cos.setObject(ros.getObject());

                  TimeSeries tempRosTimeseries = new TimeSeries();
                  tempRosTimeseries.assignTimeSeries(ros.getTimeSeries());
                  cos.setTimeSeries(clean(tempRosTimeseries));

                  for (Tuple tuple : tempRosTimeseries.getBody())
                     for (ArrayList<Record> cell : tuple.getData())
                        for (Record record : cell)
                           record.setObserverID(observerID);

                  cos.setStatus(Status.Undef);
                  ros.setStatus(Status.Cleaned);

                  if
                      (Globals.cleanedOSs.get(observerID).containsKey(ros.getObject().getID()))
                     Globals.cleanedOSs.get(observerID).get(cos.getObject().getID()).add(cos);
                  else {
                     ArrayList<ObservationSegment> cosList = new
                         ArrayList<ObservationSegment>();
                     cosList.add(cos);
```

```
                    Globals.cleanedOSs.get(observerID).put(cos.getObject().getID(),
                        cosList);
                }
            }
        }
    }
  }
}

}
```

A.4.2 AffiliateObservations

Figure A.4: Affiliate Observations Class Diagram

```
public class AffiliateObservations {

   public void MergedOSs(String observerID) throws IOException {
      if (!Globals.cleanedOSs.containsKey(observerID))
         throw new IOException("Affiliate Observations Error: The Observer ID, " +
             observerID + ", is NOT valid.");
      if (!Globals.mergedOSs.containsKey(observerID))
         throw new IOException("Affiliate Observations Error: The Observer ID, " +
             observerID + ", is NOT valid.");

      if (!(Globals.cleanedOSs.get(observerID).isEmpty())) {
         HashSet<Long> observerObjectSet = new HashSet<Long>();
         observerObjectSet.addAll(Globals.cleanedOSs.get(observerID).keySet());

         for (Long object : observerObjectSet) {
            for (ObservationSegment cos : Globals.cleanedOSs.get(observerID).get(object)) {
               if (!(cos.getStatus().equals(Status.Merged))) {
                  ObservationSegment mos = new ObservationSegment();

                  mos.setOwner(observerID);
                  mos.setObject(cos.getObject());

                  TimeSeries tempCosTimeseries = new TimeSeries();
                  tempCosTimeseries.assignTimeSeries(cos.getTimeSeries());

                  for (ObservationSegment mergingCos :
                      Globals.cleanedOSs.get(observerID).get(object)) {
                     if ((!(mergingCos.equals(cos))) &&
                         (!(mergingCos.getStatus().equals(Status.Merged))) &&
                       (mergingCos.getTimeSeries().getTimestampSet()
                       .equals(tempCosTimeseries.getTimestampSet()))) {

                       TimeSeries tempMergingCosTimeseries = new TimeSeries();
                       tempMergingCosTimeseries.assignTimeSeries(mergingCos.getTimeSeries());

                       tempCosTimeseries.horizontalMergeWIT(tempMergingCosTimeseries);

                       mergingCos.setStatus(Status.Merged);
                     }
                  }
                  mos.setTimeSeries(tempCosTimeseries);

                  mos.setStatus(Status.Undef);
```

```
            cos.setStatus(Status.Merged);

            if (Globals.mergedOSs.get(observerID).containsKey(object))
              Globals.mergedOSs.get(observerID).get(object).add(mos);
            else {
              ArrayList<ObservationSegment> mosList = new
                   ArrayList<ObservationSegment>();
              mosList.add(mos);

              Globals.mergedOSs.get(observerID).put(object, mosList);
            }
          }
        }
      }
    }
  }

}
```

A.4.3 AssociateObservations

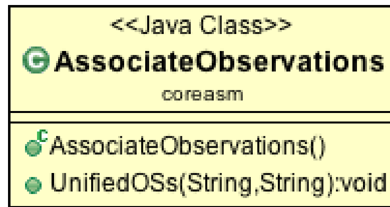


Figure A.5: Associate Observations Class Diagram

```
public class AssociateObservations {

   public void UnifiedOSs(String reasonerID, String observerID) throws IOException {
      if (!(Globals.unifiedOSs.get(reasonerID).equals(null)))
         if (!(Globals.unifiedOSs.get(reasonerID).containsKey(observerID)))
            Globals.unifiedOSs.get(reasonerID).put(observerID, new HashMap<Long,
                ArrayList<ObservationSegment>>());

      if (!Globals.mergedOSs.containsKey(observerID))
         throw new IOException("Associate Observations Error: The Observer ID, " +
             observerID + ", is NOT valid.");
      if (!Globals.unifiedOSs.containsKey(reasonerID))
         throw new IOException("Associate Observations Error: The Reasoner ID, " +
             reasonerID + ", is NOT valid.");

      if (!(Globals.mergedOSs.get(observerID).isEmpty())) {
         HashSet<Long> observerObjectSet = new HashSet<Long>();
         observerObjectSet.addAll(Globals.mergedOSs.get(observerID).keySet());

         for (Long object : observerObjectSet) {
            for (ObservationSegment mos : Globals.mergedOSs.get(observerID).get(object)) {
               if (!(mos.getStatus().equals(Status.Unified))) {
                  if
                      (Globals.unifiedOSs.get(reasonerID).get(observerID).containsKey(object))
                      {
                     for (ObservationSegment uos :
                         Globals.unifiedOSs.get(reasonerID).get(observerID).get(object)) {
                        if
                            (mos.getTimeSeries().getHead().equals(uos.getTimeSeries().getHead()))
                            {

                           TimeSeries tempMosTimeseries = new TimeSeries();
                           tempMosTimeseries.assignTimeSeries(mos.getTimeSeries());

                           uos.getTimeSeries().verticalMergeWIH(tempMosTimeseries);

                           for (int tuplei = ((uos.getTimeSeries().getBody().size()) -
                               (tempMosTimeseries.getBody().size())); tuplei <
                               uos.getTimeSeries().getBody().size(); tuplei++)
                              for (int celli = 0; celli <
                                  uos.getTimeSeries().getBody().get(tuplei).getData().size();
                                  celli++)
                                 for (int recordi = 0; recordi <
                                     uos.getTimeSeries().getBody().get(tuplei).getData()
```

```
                      .get(celli).size(); recordi++)
                        uos.getTimeSeries().getBody().get(tuplei).getData().get(celli)
                        .get(recordi).setReasonerID(reasonerID);

                  uos.setStatus(Status.Undef);
                  mos.setStatus(Status.Unified);
                }
              }
            }
            else {
              ObservationSegment newUos = new ObservationSegment();

              newUos.setOwner(reasonerID);
              newUos.setObject(mos.getObject());

              TimeSeries tempMosTimeseries = new TimeSeries();
              tempMosTimeseries.assignTimeSeries(mos.getTimeSeries());
              newUos.setTimeSeries(tempMosTimeseries);

              for (Tuple tuple : tempMosTimeseries.getBody())
                for (ArrayList<Record> cell : tuple.getData())
                  for (Record record : cell)
                    record.setReasonerID(reasonerID);

              newUos.setStatus(Status.Undef);
              mos.setStatus(Status.Unified);

              ArrayList<ObservationSegment> uosList = new
                  ArrayList<ObservationSegment>();
              uosList.add(newUos);

              Globals.unifiedOSs.get(reasonerID).get(observerID).put(newUos.getObject()
              .getID(), uosList);
            }
          }
        }
      }
    }
  }

}
```

A.4.4 CombineObservations

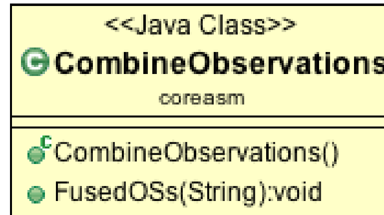


Figure A.6: Combine Observations Class Diagram

```
public class CombineObservations {

   public void FusedOSs(String reasonerID) throws IOException {
      if (!Globals.unifiedOSs.containsKey(reasonerID))
         throw new IOException("Combine Observations Error: The Reasoner ID, " + reasonerID
             + ", is NOT valid.");
      if (!Globals.fusedOSs.containsKey(reasonerID))
         throw new IOException("Combine Observations Error: The Reasoner ID, " + reasonerID
             + ", is NOT valid.");

      if (!(Globals.unifiedOSs.get(reasonerID).isEmpty())) {
         HashSet<Long> reasonerObjectSet = new HashSet<Long>();

         for (String observer : Globals.unifiedOSs.get(reasonerID).keySet())
            reasonerObjectSet.addAll(Globals.unifiedOSs.get(reasonerID).get(observer).keySet());

         for (Long object : reasonerObjectSet)
            if (!(Globals.endingTimestamp.get(reasonerID).containsKey(object)))
               Globals.endingTimestamp.get(reasonerID).put(object, new HashMap<String,
                   Date>());

         if (!(reasonerObjectSet.isEmpty())) {
            for (Long object : reasonerObjectSet) {
               if (Globals.fusedOSs.get(reasonerID).containsKey(object)) {
                  for (String observer : Globals.unifiedOSs.get(reasonerID).keySet()) {
                     if
                         (Globals.unifiedOSs.get(reasonerID).get(observer).containsKey(object))
                         {
                        for (ObservationSegment uos :
                            Globals.unifiedOSs.get(reasonerID).get(observer).get(object)) {
                           if(!(uos.getStatus().equals(Status.Fused))) {

                              TimeSeries tempUosTimeseries = new TimeSeries();
                              tempUosTimeseries.assignTimeSeries(uos.getTimeSeries());
                              tempUosTimeseries
                              .splitTimeSeriesFromTimestamp(Globals.endingTimestamp.get(reasonerID)
                              .get(object).get(observer));
                              Globals.fusedOSs.get(reasonerID).get(object).getTimeSeries()
                              .verticalMergeWIH(tempUosTimeseries);

                              Globals.fusedOSs.get(reasonerID).get(object).setStatus(Status.Undef);
                              uos.setStatus(Status.Fused);
```

```
                Globals.endingTimestamp.get(reasonerID).get(object).put(observer,
                    tempUosTimeseries
                .getEndTimestamp());

            }
         }
      }
   }
}
else {
   ObservationSegment newFos = new ObservationSegment();

   newFos.setOwner(reasonerID);

   for (String observer : Globals.unifiedOSs.get(reasonerID).keySet()) {
      if
          (Globals.unifiedOSs.get(reasonerID).get(observer).containsKey(object))
          {
         for (ObservationSegment uos :
             Globals.unifiedOSs.get(reasonerID).get(observer).get(object)) {
            if(!(uos.getStatus().equals(Status.Fused))) {

               TimeSeries tempUosTimeseries = new TimeSeries();
               tempUosTimeseries.assignTimeSeries(uos.getTimeSeries());

               Globals.endingTimestamp.get(reasonerID).get(object).put(observer,
                   tempUosTimeseries
               .getEndTimestamp());

               for (String mergingObserver :
                   Globals.unifiedOSs.get(reasonerID).keySet()) {
                  if (Globals.unifiedOSs.get(reasonerID).get(mergingObserver)
                  .containsKey(object)) {
                     for (ObservationSegment mergingUos :
                         Globals.unifiedOSs.get(reasonerID)
                     .get(mergingObserver).get(object)) {
                        if ((!(mergingUos.getStatus().equals(Status.Fused))) &&
                            (!(mergingUos.equals(uos)))) {

                           TimeSeries tempMergingUosTimeseries = new
                               TimeSeries();
                           tempMergingUosTimeseries
                           .assignTimeSeries(mergingUos.getTimeSeries());

                           Globals.endingTimestamp.get(reasonerID).get(object)
                           .put(mergingObserver, tempMergingUosTimeseries
                           .getEndTimestamp());

                           tempUosTimeseries.verticalMergeWIH(tempMergingUosTimeseries);

                           mergingUos.setStatus(Status.Fused);
                        }
                     }
                  }
               }
               newFos.setTimeSeries(tempUosTimeseries);

               newFos.setObject(uos.getObject());
```

```
                    uos.setStatus(Status.Fused);

                    Globals.fusedOSs.get(reasonerID).put(object, newFos);
                  }
                }
              }
            }
          }
        }
      }
    }
  }

}
```

A.4.5 ResolveInconsistencies

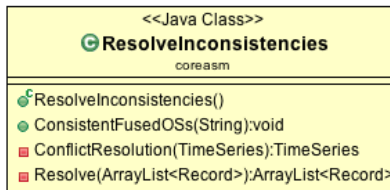


Figure A.7: Resolve Inconsistencies Class Diagram

```
public class ResolveInconsistencies {

   public void ConsistentFusedOSs(String reasonerID) throws IOException {
      if (!Globals.fusedOSs.containsKey(reasonerID))
         throw new IOException("Resolve Inconsistencies Error: The Reasoner ID, " +
             reasonerID + ", is NOT valid.");

      if (!(Globals.fusedOSs.get(reasonerID).isEmpty())) {
         for (Long object : Globals.fusedOSs.get(reasonerID).keySet()) {
            if (!(Globals.fusedOSs.get(reasonerID).get(object).getStatus()
            .equals(Status.Consistent))) {
               ObservationSegment cFos = new ObservationSegment();

               cFos.setOwner(reasonerID);
               cFos.setObject(Globals.fusedOSs.get(reasonerID).get(object).getObject());

               TimeSeries tempFosTimeSeries = new TimeSeries();
               tempFosTimeSeries.assignTimeSeries(Globals.fusedOSs.get(reasonerID)
               .get(object).getTimeSeries());

               cFos.setTimeSeries(ConflictResolution(tempFosTimeSeries));

               cFos.setStatus(Status.Undef);
               Globals.fusedOSs.get(reasonerID).get(object).setStatus(Status.Consistent);

               Globals.cfusedOSs.get(reasonerID).put(object, cFos);
            }
         }
      }
   }

}
```