A PROTOTYPE IMPLEMENTATION FOR SITUATION ANALYSIS USING ASP AND COREASM

by

Zahra Vaseqi

B.Sc., Amirkabir University of Technology, 2009

A 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

SIMON FRASER UNIVERSITY
Summer 2012

APPROVAL

Name:	Zahra Vaseqi
Degree:	Master of Science
Title of Thesis:	A Prototype Implementation for Situation Analysis using ASP and CoreASM
Examining Committee:	Dr. Petra Berenbrink Chair
	Dr. Jim Delgrande, Senior Supervisor
	Dr. Uwe Glässer, Supervisor
	Dr. Fred Popowich, SFU Examiner
Date Approved:	August 30, 2012

Partial Copyright Licence

Abstract

Answer Set Programming (ASP) is a non-monotonic declarative language that is widely applied to different areas in science and technology. Early implementations of ASP could only accept static input data which limited the usefulness of such programs. However, recent advancements enabled dynamic input data to be used and generate answer sets incrementally.

In this work we assess the suitability of a reactive ASP system to underpin a situational awareness program. We used the *Maritime traffic* domain for our evaluation, as it is highly dynamic with new information continuously pouring in.

We have developed a situation assessment system to automate a component of the marine traffic control system in order to assist human operators by automatically deriving new situational facts and suggest possible courses of action. The State Transition Data Fusion model has been adopted for the situation assessment task; while the *CoreASM* and ASP components perform the underlying analysis.

Keywords: Reactive Answer Set Programming, Situation Awareness, State Transition Data Fusion Model.

To my parents, Fatemeh and Mohamadtaqi,
for their endless love and support;

to my dearest friend, Zahra Nazari,
for her inspiring friendship.

Acknowledgments

I would like to give my gratitude to my supervisor Dr. Jim Delgrande for his valuable time and support. Thanks are given to Dr. Uwe Glässer for the inspiration of the application used in this project. I also would like to give my appreciation to my thesis examiner, Dr. Fred Popowich, for his valuable comments to improve my thesis.

I would also want to thank all my friends and colleagues for their valuable help; in particular: Narek Nalbandyan, Amir Aavani, Piper Jackson, Hamed Yaghoubi Shahir, Mark Roth, and Shahab Tasharrofi for being so patient to answer my many questions; and the *oclingo* developer team at the University of Potsdam who promptly answered my inquiries.

I am also grateful to my beloved family for their love and motivation.

Contents

List of Figures

List of Programs

Chapter 1

Introduction

1.1 Motivation

Technological advancements have made it possible to generate vast improvements in efficiency and automation. Recently, these advancements have been applied to information gathering and retrieval and are now causing an information overload. Many occupations require making real time decisions based on a flood of incoming information. Some examples would be airline controllers, emergency dispatch, and maritime operations. These domains [1] have a combination of a huge set of rules combined with constantly changing information. The human operators are placed under a big burden. The combinatorial explosion of how different rules, regulations, and constraints interact with each other makes it difficult for the human operator to generate a clear awareness of the current situation. Automation of human operator tasks is a key component in generating better situational awareness. As situational awareness increases, better decisions can be made.

A good model is essential for any context. A *situation awareness system* can be utilized to augment the human operator's expertise and use the available information to detect anomalous actions and events. These *situation awareness systems* should be able to provide information at a variety of abstraction levels. Take for example a maritime controller trying to determine if ships are following shipping regulations. Low level situational facts are easy to determine such as which boats are exceeding the speed limit. However, many situational

[1] Informally, "*domain*" refers to the rules used to analyze the environment as well as the information received regarding the state of the environment.

facts are not directly observable and require domain expertise and inference. For example, in order to detect suspicious vessels we may need to access low level situational facts as well as the more abstract situational facts inferred based on the information in the past.

Logic programming languages are well suited for the higher level situational analysis as they provide an intuitive way of expressing rules and information. In the past, logic programming was not well equipped to handle problem domains that had a time (dynamic) component, which is a necessity in a situational awareness system. However, recent advancements in some logic implementations have allowed for the ability to handle dynamicity in a seamless way. This led us on the path to investigate the benefits and potential issues using a reasoning system within a *situational assessment system.*

1.2 Objectives

As stated in the previous section, we wish to investigate using a logical reasoning system within *situational awareness* application. However, there are many kinds of logical reasoning systems to choose from. For this paper, we chose to use Answer Set Programming (ASP), the specifics of which will be discussed in Section 2.1.

ASP has a number of interesting features which makes it desirable to use for *situational awareness.* It has a rich potential in representing rule-based domains because it is declarative and provides a compact and intuitive encoding of the domain expert's knowledge within a non-monotonic framework. The non-monotonic nature of answer set programming makes it rather suitable for dynamic domains as it provides a concise way of representing default rules, and preferences over choices. Moreover, real world rule-based domains are usually prone to policy (rule) changes which ASP is flexible enough to extend and can adapt the system as the policies change. Recent advancements to implementations of ASP enable incremental evaluation of logic programs; this allows for the program to have a notion of history which is essential for a *situational awareness* system. The *history* of the system includes the raw information received from the environment as well as the knowledge inferred in the system. The history in the system grows throughout the time and new situational facts are inferred. In order to handle the increasing information accumulating over time, efficient means to store the history will be beneficial. Furthermore, current ASP implementations[2]

[2]The ASP-solver used in this work takes Normal Logic Programs as its input language which is expressive

are powerful enough to express problems from NP-complete complexity class with a concise and intuitive encoding. Refer to [19] for an example.

The primary objective of this work is to assess the suitability of ASP as a declarative logic programming language for high-level situational analysis. This evaluation consists of an analytical analysis which involves an extensive review of ASP itself and the trade-offs of using it within a *situational awareness* system.

To provide further analysis of the practical considerations involved with building a real time *situational analysis* system, we constructed an implementation of a situational awareness system targeted for the *Marine Traffic Control* domain. This implementation was created by combining an existing logical system, *oclingo*, into a multi-layered *CoreASM* model. The design rationale will be discussed in Chapter 3.

Ultimately, our goal is to advance the capabilities of real time *situational awareness* systems by integrating a reactive ASP solver.

1.3 Related Work

In this section we will discuss several recent papers relating to situational awareness. Each of the works uses a multi-layered approach in addressing the problem of situational awareness. The selection of components used for each layer has various strengths and weaknesses which we will discuss in detail.

The architecture for situation awareness systems (SA) proposed by Baader et al. [1] satisfies the requirements to be situationally aware as defined by Endsley[5]. Baader's architecture contains three levels of analysis: *data aggregation*, *semantic analysis* and *alert generation* corresponding to the three main components of Endsley's definition, namely: *perception*, *comprehension* and *projection*.

The data aggregation layer in this architecture is responsible for perception of elements by monitoring data sources. It identifies entities along with their low-level properties. The semantic analysis layer comprehends and evaluates entities in conjunction with background knowledge producing an understanding of the overall meaning of the identified entities. In this layered architecture, the system tries to realize the relations between the entities identified in the perception layer and draws out the situational facts that need more abstract

enough for encoding problems in the NP-complete complexity class. While some of the other available ASP-solvers accept Disjunctive Logic Programs that are more expressive compared to Normal Logic Programs.

analysis in the semantic analysis layer. Finally, the alert generation layer generates an appropriate alert where the situational facts have the potential to correspond to an anomalous event and require a human analyst's attention.

Baader et al.[1] use an ASP formalism underlying a first order logic reasoner in addition to data fusion techniques for the data aggregation layer, and a description logics (DL) inference engine for the semantic analysis. The DL vocabulary include three basic building blocks: *individuals* (constants), *roles* (denoting binary relationships between individuals), and *concepts* (denoting groups of individuals). A knowledge base in description logics is composed of two components:

- a TBox or *terminological* box containing relations among concepts and rules;
- an ABox or *assertion* box containing assertions on individuals.

The ABox component of the description logic knowledge base in Baader et al.[1] is mainly formed in the data aggregation layer; while the TBox component contains conceptual background information in terms of rules shaping the semantic layer of the model. The ABox and the TBox are linked through their common terminology and higher-level situational facts are inferred using a DL reasoner.

One of the main strengths of this approach comes from DL languages being highly expressive and powerful to encode the rules. It also facilitates representing object categories in the system. However, it may not be as powerful as reactive ASP in tracking the history of objects and events. In this thesis, we use the State Transition Data Fusion (STDF)[17] model for situation awareness which shares the high-level components of the model by Baader et al. STDF provides a hierarchal framework which provides modularization and delegation of tasks to programs that are best suited to handle each component. In the semantic analysis module, we replaced the DL inference engine and divided the module into two components. These two components are the *CoreASM* modelling framework and a reactive ASP-solver. The *CoreASM* performs part of this task in an imperative fashion, while the reactive ASP-solver performs the higher-level task declaratively. The reactive ASP-solver also provides the ability to handle the history of the domain inside the module.

Mileo et al.[22] use a logic-based approach in an application to support elderly patients' well-being at home. Their system provides a context-aware setting to interpret sensory signals from a house monitoring system in order to explain and predict the patient's health status. They develop a logic-based situation assessment model to map the input sensory

data into meaningful ASP predicates. The domain rules including the rules to analyze the environment as well as the rules to interpret a patient's situation are encoded using ASP. An ASP-solver then uses the interpreted sensory signals and the inference rules to reason on patient-specific needs as well as the evolving state of the patient in the environment. This work is similar to ours in that the application of choice is taken from a dynamic domain and a declarative approach is applied. However, there are two major differences, the first being that we use STDF for our situational awareness system. The STDF model, as a generic and domain-independent model, allows us to clearly delegate each of the tasks to an appropriate module to handle. The second difference is how the dynamic aspect of the domain is handled. They just associate a time component to the dynamic predicates; however, their model requires passing the history to future time steps as their ASP solver does not provide a means to handle the history of the domain.

Roy [25] develops a rule-based expert system composed of an inference engine which takes low-level situational facts and generates additional facts based on them. It represents an expert's knowledge in terms of *IF-THEN* rules. At each step the *IF* part of a rule gets matched against asserted facts, and the *THEN* part of the rule will be added to the facts whenever a positive match occurs. The syntax of the rules are as follows:

$$IF\ [fact_1\ and\ fact_2\ and\ \ldots\ fact_n]\ THEN\ fact_x$$

The ASP-based approach taken in the present paper provides a richer syntax to represent rules in the system. As an example, ASP default rules defined using the notion of *negation-as-failure* enable a concise encoding of the non-changing properties of the domain, which is not representable given the above syntax. Furthermore, incremental answer set generation makes it easier to keep track of the history of objects and events in the environment.

Nogueira et al.[23] use ASP for an application in public health data analysis. They first present a semantic model for the food safety domain in which they include events that can be considered as warnings for a potential foodborne illness outbreak together with relationships between the components that describe each event. Their system, called NCFEDA[1], follows a similar architecture to that of Baader et al.[1], and is composed of three components: an *event manager*, gathering information from various sources; a *semantics module*, containing background conceptual information and rules describing the domain; a *rule-based inference engine* which is an ASP-solver. All the rules, facts, and ontologies –

[1]North Carolina Foodborne Events Data Integration and Analysis Tool

describing foodborne diseases and their related syndromes – in the system are encoded as ASP rules which facilitates inference of new factual information and deducing relationships among events.

Farahbod et al.[7] proposes a situation analysis approach based on abstract state machines implemented using the *CoreASM* modelling framework. Their model takes a number of autonomous dynamic agents interacting with each other and the environment by manipulating the machine state using the transition rules defined in their programs. A distinguished observer agent is defined to perform the situation analysis task based on the STDF model where a *rendezvous detection* task in marine domain is to be performed.

So far we have reviewed several situational awareness systems and identified some of their undesirable aspects. One of the common deficits shared by some these systems is the poor handling of notion of time. History of the domain plays a key role in obtaining the state of situation awareness and having a clear way of handling the history is very important. In our work, we attempt to address some of these short comings by using a reactive ASP solver to handle the history of the domain in a seamless way. One of the remarkable features of this work is the use of a reactive answer set solver which gives us more flexibility in dealing with knowledge from the previous time steps.

1.4 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 provides a review of the underlying concepts including Answer Set Programming (ASP), Situation Awareness, State Transition Data Fusion (STDF) model, and the CoreASM modelling framework.

In **Chapter 3**, we first present a description of the *Marine Traffic* domain which is adopted as an application to illustrate the situation analysis task. An account of the customized version of the STDF model is presented followed by an explanation of the implemented model components, using *CoreASM* modelling framework.

Chapter 4 explains how high-level situation analysis can benefit from ASP and the facilities made available by one of its recent reactive implementations.

Lastly, **Chapter 5** discusses the strengths and shortcomings of the approach. This chapter concludes with a summary of this thesis along with some tracks for future work.

Chapter 2

Background

2.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative language introduced by Gelfond and Liftchitz[15] in the late 90's. Like other declarative languages, ASP requires the programmer to describe the problem by stating *what* the problem domain is like and *what* is to be computed, but not *how* to compute it. The declarative paradigm takes the burden of how to compute the solution. In contrast to this is the imperative paradigm in which the programmer must provide an algorithm to compute the solution to a problem. There can be multiple solutions to a problem; these solutions are known as the answer sets. Logic programming with stable model semantics or so called Answer Set Programming (ASP), ASP enables the programmer to view a problem in a higher level of abstraction with a clear non-monotonic semantic model.

In ASP a given problem is represented as a *logic program* and the solution(s) correspond to the resulting *answer sets*. An ASP system usually consists of a grounder and a solver [14], where the grounder takes an ASP program, that can contain variables, as its input and translates it into a propositional variable-free program. The resulting grounded program then gets passed to the solver to generate answer sets of the program[1]. An ASP program is a logic program which is a collection of statements analogous to *if-then* rules. These rules have the form:

$$< \texttt{head} > \ :- \ < \texttt{body} >$$

[1]Please see [14] for a detailed explanation

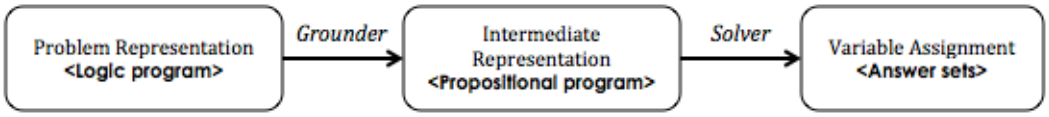

Figure 2.1: ASP Systems Architecture [14]

Where the *body* represents the *if* part, the *head* represents the *then* part. The *body* of the rule can contain a conjunction of literals, i.e. negated and non-negated atoms, as well as literals with negation as failure. The *body* or the *head* of a rule can be empty; when the body is empty the *head* represents a *fact*. While the rules with empty *head*s represent *constraint*s.

After the grounder translates the program into a variable-free, grounded propositional program, the solver interprets the program under stable models semantics, which is a non-monotonic semantic, and generates the answer sets. A program may have zero or more answer sets where each answer set contains a subset of heads of the rules that can be inferred given the answer set.

A more detailed explanation of ASP systems components based on a recent implementation of ASP, *clingo*[2], and one of its extensions *oclingo*, will be discussed in this section.

The remainder of this section is organized as follows: Section 2.1.1 describes ASP language syntax and process of grounding the input program, Section 2.1.2 explains answer set semantics and the process of generating answer sets, Section 2.1.3 will then describe *oclingo*, the reactive answer set solver, and its input language. *Oclingo* is an incremental logic program for dealing with dynamic information. Finally Section 2.1.4 describes the general methodology in writing ASP programs.

[2]We have based our work on the ASP implementation by University of Potsdam, *clingo*, since the main focus of this thesis is on applying ASP to dynamic domains. To the best of our knowledge, the recent extensions to *clingo*, namely *iclingo* and *oclingo* are the only available implementations that enable dealing with dynamic domains. However, the ASP semantics is in common between all the implementations and there are minor syntactical differences between different implementations.

2.1.1 ASP Language Syntax

In this section we will briefly describe the basic elements of normal logic programs and explain the *clingo* language syntax.

Building blocks of ASP programs are as listed below:

- a set C of **constants**.
- a set V of **variables**.
- a set F of **functions** of the form $f(t_1, \ldots, t_n)$ where f is the function name with n parameters and where each of the parameters, t_i, is called a *term*.
- a set P of **predicates** of the form $p(t_1, \ldots, t_n)$ where p is the predicate name taking n terms t_i, for $i \in \{1, \ldots, n\}$, as parameters. A predicate that does not contain any variable parameters is called an *atom*.

Each function, variable, and constant is considered to be a **term**. A function takes terms as its parameters; therefore terms are defined recursively and can be nested. Predicates in the language can appear in negated[3] or non-negated form and they are called *literals*. There are generally two types of negations defined in ASP; *strong negation*, denoted by '¬', and *default negation*, denoted by '`not`'. *Default negation* is also referred to as *negation-as-failure-to-prove* (naf), as '`not atom`' holds only if there is no evidence to prove the truth of the `atom`. The concept of *default negation* brings in non-monotonicity into ASP and allows retraction of previous conclusions as new information comes in. This notion remarkably enhances its knowledge representational aspect and enables a compact representation of defaults, representing qualitative preferences, and transitive closure. Later in chapter 4, we will further explain the representational aspect of ASP.

The ASP systems take logic programs (a.k.a ASP programs) as their input. These logic programs are a set of rules of the form:

$$\mathtt{A_0} \ :- \ \mathtt{A_1}, \ldots, \ \mathtt{A_m}, \ \mathtt{not\ A_{m+1}}, \ldots, \ \mathtt{not\ A_n}. \tag{2.1}$$

where $\mathtt{A_0}$ is the head of the rule and $\mathtt{A_1}, \ldots, \ \mathtt{A_m}, \mathtt{not\ A_{m+1}}, \ldots, \ \mathtt{not\ A_n}$ is the body of the rule. Logic programming rules function as inference rules where intuitively $\mathtt{A_0}$ is inferred if

[3] using *strong* negation, '¬'

$\texttt{A}_1, \ldots, \texttt{A}_\texttt{m}$ hold and there is no evidence supporting any of $\texttt{A}_{\texttt{m}+1}, ..., \texttt{A}_\texttt{n}$. The inferred atom, $\texttt{A}_0$, will no longer hold once we acquire evidence supporting at least one of the naf-literals in the body of the rule. In the rest of this thesis, $\texttt{A}_1, \ldots, \texttt{A}_\texttt{m}$ are called positive literals[4], and $\texttt{not A}_{\texttt{m}+1}, \ldots, \texttt{not A}_\texttt{n}$ are called naf-literals.

There are three basic type of rules defined in ASP, namely *regular rules*, *facts*, and *integrity constraints*.

```
threat(X,Y,t) :-
     danger_combined_cargos(X,Y,t),
     close_distance(X,Y,t),
     vessel(X,t), vessel(Y,t).
```

The rule above is an example of a **regular rule** used to describe the situational fact `threat` caused by vessels `X` and `Y` at time point `t`. This rule draws a situation `threat` if two vessels are in close proximity carrying cargos that combined together would be dangerous.

Facts are rules with an empty body of the form (2.2). They can be used to represent the information in the domain, where truth of the atom is not conditional on any other atoms.

$$\texttt{A}_0. \tag{2.2}$$

As an illustration, $\texttt{vessel}(\texttt{ID}, \texttt{LOCATION}, \texttt{T})$. is an example of a fact where we know at time point `T` there exists a vessel with specified `ID` at location `LOCATION`.

Lastly, **integrity constraints** are rules with empty heads. They prohibit co-occurrence of literals in the body of the rule in one answer set. Adding integrity constraints to the program may omit some of the answer sets. Integrity constraints, in general, are in the following form:

$$:- \texttt{A}_1, \ldots, \texttt{A}_\texttt{m}, \texttt{not A}_{\texttt{m}+1}, \ldots, \texttt{not A}_\texttt{n}. \tag{2.3}$$

Below is an example of an integrity constraint where we would want to restrict the presence of two ships in the same shipping lane at the same time point.

$$:-\texttt{lane\_occupied}(\texttt{LANE}_1, \texttt{VID}_1, \texttt{T}), \texttt{lane\_occupied}(\texttt{LANE}_1, \texttt{VID}_2, \texttt{T}), \texttt{VID}_1 != \texttt{VID}_2.$$

[4]Positive literals can be negated atoms using classical negation. ($\neg atom$)

All ASP implementations have share common basic language constructs. However, each implementation can have additional language features that may be specific to that implementation or shared with some but not all implementations. Some of *oclingo* language features used in subsequent chapters are listed below.

- **Range Operator**, `time(1..100).` is an example of using range operator in the head of a rule. The grounding process will then generate all the following rules to be used by the solver: `time(1).` `time(2).` ..., `time(99)`, `time(100)`
- **Arithmetic Expressions**, '+', '−', '/', '∗', '`mod`', etc.
- **Relational Operators**, '<', '>', '<=', '>=', '==', '!='.
- **Classical Negation**, '-'.
- **Pooling**, ';', allows rules to be encoded more compactly. The atom $p(..., X; Y, ...)$ is the shorthand for two options: $p(..., X, ...)$ and $p(..., Y, ...)$. When appeared in the body of a rule, the pooled atom is expanded to a conjunction of the options within the same body.
- **Domain Expander**, ':', takes an atom with variable(s) as its left hand side operand and a domain(s) for the variables as its right hand side operand. It expands the left hand side operand variable(s) over its domain(s).
- **Aggregates**, *clingo* supports a handful number of aggregates, including `count`, `sum`, `avg`, `even`, `min`, and etc. `count` and `sum` aggregates are explained below as they have been widely used in our work. The **count** aggregate can be used with the syntax

$$\texttt{lower}\ \#\texttt{count}\{\texttt{atom}_1, \ldots,\ \texttt{atom}_n\}\ \texttt{upper}^5 \tag{2.4}$$

  where *lower* and *upper* are integers, and a set of atoms are braced inside a pair of curly brackets. The expression evaluates to true if the number of true atoms in the curly brackets is between the lower and upper bound. This is also called a *cardinality constraint*. The following displays a *choice rules* which is modelled using `count` aggregate:

$$\texttt{lower}\ \{\texttt{a}_0,\ \ldots,\ \texttt{a}_k\}\ \texttt{upper} :- \texttt{A}_1,\ \ldots,\ \texttt{A}_m,\ \texttt{not}\ \texttt{A}_{m+1},\ \ldots,\ \texttt{not}\ \texttt{A}_n. \tag{2.5}$$

[5] `#count` keyword is optional.

A choice rule expects $\texttt{lower} \leq \texttt{i} \leq \texttt{upper}$ number of braced atoms in the head get a truth value if the body of the rule holds. The **sum** aggregate is defined similarly except that it uses a multi-set[6] of atoms with an optional weight[7] parameter for each member.

$$\texttt{lower}\ \#\texttt{sum}[\texttt{atom}_1 = \texttt{w}_1, \ldots,\ \texttt{atom}_n = \texttt{w}_n]\ \texttt{upper}^{8} \tag{2.6}$$

Alternatively the `lower` and `upper` bounds can be omitted and the result of the summation assigned to a variable, i.e.

$$\texttt{S} = \#\texttt{sum}[\texttt{atom}_1 = \texttt{w}_1, \ldots,\ \texttt{atom}_n = \texttt{w}_n] \tag{2.7}$$

- **Optimization Statements** enable generation of optimized answer sets based on some criteria. An optimization statement can either be a minimization or a maximization statement. As shown below, an optimization statement takes the criteria for optimization braced inside a pair of square brackets and also a number indicating the priority of criteria in case of more than one criterion:

$$\#\texttt{minimize}[\texttt{atom}_1 = \texttt{w}_1\ @\ \texttt{p}_1, \ldots,\ \texttt{atom}_n = \texttt{w}_n\ @\ \texttt{p}_n] \tag{2.8}$$

where $\texttt{p}_\texttt{i}$ for $i \in 1, \ldots, n$ is an integer number indicating the priority of the criteria. Roughly speaking, an answer set is optimal if the sum of weights of literals that hold is maximal or minimal, as indicated by the statement, among all answer sets of the given program [14].

2.1.2 Answer Sets and Stable Model Semantics

The stable model semantics, or the answer set semantics, is one of the approaches to the meaning of negation as failure which forms the basis for answer set programming. Intuitively, an *answer set* is a set of atoms S, which is a subset of head atoms in the rules, where the head of a rule belongs to S only if all the positive literals of the body of the rule belong to S and none of its naf-literals appear in S. One of the important remarks in generating answer sets is that answer sets are defined in terms of atoms that are actually variable-free

[6] Repeating atoms with the same or different weights are permitted.

[7] If not indicated, the default value for the weight(s) would be 1.

[8] `#sum` keyword is optional.

predicates. Therefore, the logic program must be grounded into its equivalent variable-free version through the grounding process. The grounding process replaces every occurrence of the variables in the program with all the possible substitutions from the constants and the functions in the program. The solver then operates on the ground program to generate the answer set(s).

In order to explain the answer set generation process, we start with the naf-literal free programs where there is no occurrence of negation-as-failure. The rules in a naf-literal free program have the following form:

$$\mathtt{A_0} \ :- \ \mathtt{A_1}, \ldots, \ \mathtt{A_m}. \tag{2.9}$$

Such programs are called naf-free logic programs and only have one unique answer set. For example, assume the following naf-free logic program from [15]:
$\mathtt{p}(1,2)$.
$\mathtt{q}(\mathtt{x}) :- \mathtt{p}(\mathtt{x},\mathtt{y}), \neg\mathtt{q}(\mathtt{y})$.
The grounding process turns the program into the following variable-free program:
$\mathtt{p}(1,2)$.
$\mathtt{q}(1) :- \mathtt{p}(1,1), \neg\mathtt{q}(1)$.
$\mathtt{q}(2) :- \mathtt{p}(2,1), \neg\mathtt{q}(1)$.
$\mathtt{q}(1) :- \mathtt{p}(1,2), \neg\mathtt{q}(2)$.
$\mathtt{q}(2) :- \mathtt{p}(2,2), \neg\mathtt{q}(2)$.
In order to find the answer set in a naf-free program we need to start with a subset of atoms in the program called S and plug them into the bodies of the rules to infer new atoms, if possible[9]. An iterative process of inferring new atoms is then performed; at each iteration we form a set of atoms S^t corresponding to the set of inferrable atoms at the current iteration. After a finite number of iterations [10] the set S^t stops evolving; in other words, the set $S^t = S^{t-1}$. The set S is called a stable model or an answer set, if the final set S^t is equal to the initial set S. In the above example the only answer set is $\{\mathtt{p}(1,2)\}$ as we can not infer any other head atoms except for the given fact $\mathtt{p}(1,2)$.

Extending the above algorithm to the generic logic programs containing naf-literals requires a strategy to deal with the naf-literals. The main issue with naf-literals is that at

[9]The initial set S can be an empty set.

[10]The answer set generation process is guaranteed to terminate as ASP limits the language to the predicates and functions with a finite domain

the start of the inference phase we do not know if they are derivable or not, and therefore we need to make assumptions about their status. As explained earlier, regular logic programs are composed of the rules of the form:

$$\mathtt{A_0} \ :- \ \mathtt{A_1}, \ldots, \ \mathtt{A_m}, \ \mathtt{not\ A_{m+1}}, \ldots, \ \mathtt{not\ A_n}. \tag{2.10}$$

In this general case we need to make an assumption regarding the status of each of the naf-literals, '$\mathtt{not\ A_k}$', guessing which of them are derivable in the program. If $\mathtt{A_k}$ is guessed to be derivable in the program, '$\mathtt{not\ A_k}$' is assumed *false*; otherwise, '$\mathtt{not\ A_k}$' is assumed *true*. We will then need to verify each of the guesses by replacing the assumptions into the original program. Replacing corresponding *true* and *false* values in the program turns the program into a naf-free logic program. In the verification phase if our guess was compatible we would get an answer set that includes the naf-literals that were assumed to be false, as they were assumed to be derivable, and does not include the ones assumed to be true. The fact that for a given arbitrary program we may have more than one set of guesses on derivability of the naf-literals leads to possibility of having multiple answer sets corresponding to multiple solutions for a problem. Apparently this non-determinism is also a consequence of having negation-as-failure[11]. As an example, assume the following logic program with two ground rules:

$\mathtt{p} :- \mathtt{not\ q}.$

$\mathtt{q} :- \mathtt{not\ p}.$

$\mathtt{w} :- \mathtt{q}.$

Let's start with the set $S = \{p, w\}$; assuming that $\mathtt{p}$ is true leads to falsifying $\mathtt{not\ p}$, and therefore the rule "$\mathtt{q} :- \mathtt{not\ p}$" would not be of any use as its body does not hold given the set S. The set S does not give us any information about the atom $\mathtt{q}$ so we can assume truth value for the naf-literal "$\mathtt{not\ q}$"; as a consequence the rule "$\mathtt{p} :- \mathtt{not\ q}$" will turn into a fact as its body gets truth value. As shown below, after applying the assumptions in the original program we get a naf-free program:

$\mathtt{p}.$

$\mathtt{w} :- \mathtt{q}.$

Given the naf-free program, the only atom that can be inferred is the atom p. Therefore the

[11]The non-determinism in answer set programming that usually results in multiple distinct answer sets originates in several properties of the language. As we described above, the notion of negation-as-failure is one of its sources. Choice rules, described in Section 2.1.1, and disjunction operator that are defined in Disjunctive Logic Programs are some of the other sources of such non-determinism in ASP.

inferred atoms set S^1 after one iteration would only include atom p. Since the only other rule in the naf-free does not infer any new atoms the inference process terminates. Since $S \neq S^1$, S is not a stable model.

Let's run through the previous example using a different initial set $S = \{q, w\}$. This set does not contain atom p; therefore we must assume truth value for `not p`. However, the atom q is included in the initial set so the `not q` is falsified. The resulting naf-free program is presented below:

```
q.
w :- q.
```

The only atom that can be inferred in the first run through the naf-free program is q, and thus we get $S^1 = \{q\}$. Running through the program for a second iteration, we can infer atom w using the inference rule "`w :- q`". Therefore we get $S^2 = \{q, w\}$. The third iteration through the program does not infer any new atoms; therefore the answer set generation process terminates. The resulting set S^2 contains the same set of atoms as the initial set S; therefore it is a stable model for the original program.

Gelfond and Liftchitz [15] were the first to formally define stable model semantics for basic logic programs; Simons [27] then extended the semantics for more expressive logic program rules including constraints, choice, and weight rules. In this section we provide the formal definition for programs with regular rules, facts, and constraint rules.

The *reduct* of the program P given an initial set S, which is denoted by P^S produces a naf-free program where we can verify if the initial set S is a valid answer set for program P. The following steps summarize the process of generating the P^S:

1. deleting all regular rules in P where there is a negated literal, `not s`, in its body such that $s \in S$
2. deleting all negated literals, `not a`, from the remaining rules.

Step (1) in the above procedure ensures that P^S would not be able to infer atoms through rules where there exists a naf-literal belonging to S that appears in the body. Step (2) removes all the negated literals from the body of the rest of the rules as there is no information to either prove of disprove them. Stated more concisely, reduct of program P with respect to the set S prepares the program to only infer valid atoms.

Definition 1. A set of atoms S is a stable model of program P if and only if S is the

deductive closure of P^S when the rules in P^S are seen as inference rules.

This definition ensures that all the atoms in the set S are deducible, and that they are the smallest set of atoms deductible. Being smallest is implied by the fact that if the initial set S shrinks through the process, it would not be called an answer set as it is giving truth values to some atoms without having the program support them.

In order to extend the above definition to the programs with constraint rules, we need to ignore set S if its atoms violate any of the constraint rules. Therefore an answer set is the smallest set of atoms that can be inferred by the program, and does not violate any of the constraint rules.

In summary, the overall task of the solver would be to pick a potential answer set, S, and follow the above procedure to confirm that S is a valid answer set.

2.1.3 Reactive Answer Set Programming

Reactive answer set programming bridges the gap between the declarative paradigm used by answer set programming and a wide variety of applications dealing with dynamic domains. Gebser et al. [12] introduce this approach by incorporating online data streams into the logic program. They take advantage of incremental logic programs[13] to bring in dynamicity through a time component. The reactive ASP-solver augments the incremental logic program by enabling the solver to use additional external atoms coming from outside of the logic program. Time steps from the dynamic domain are aligned to the steps of the incremental logic program to admit additional grounded atoms into the solver. This useful extension to incremental answer set programming makes a connection from the dynamic domain to the solver. The rest of this section presents a description of *incremental logic programs* which is the input language for *oclingo*, currently the only available reactive answer set solver. In the subsequent section, we will look at how *oclingo* works.

Incremental Logic Programs

Incremental logic programs are a type of normal logic program augmented by an additional dynamic parameter. This dynamic parameter can be aligned to the time steps and assumed to be a time parameter. This changing parameter enables augmenting the ground logic program rules with permanently added as well as transient new rules that emerge as time

Program 2.1: Incremental Logic Program

```
1       #external predicate_1/arity_1.
2       #base.
3       ...
4       #cumulative t.
5       ...
6       #volatile t:duration_1.
7       ...
8       #volatile t:duration_2.
9       ...
```

goes on. This concept is introduced by Gebser et al [13] and has been used as a basis to implement the concept of reactive answer set programming[12].

The incremental logic program has been used as the input language for the reactive answer set solver, *oclingo*. This input language shares all the fundamental concepts and the basic language constructs with the introduced ASP syntax for *clingo*. The only major difference is that incremental logic programs are composed of three logic programs, namely *base*, *cumulative*, and *volatile* programs. The `base` program is time independent, while the other two parts are defined to capture the notion of time.

1. `base` program — includes static knowledge, independent of parameter t;
2. `cumulative` program — includes knowledge accumulating with increasing t;
3. `volatile` program — includes time-decaying knowledge.

The `base` program includes static knowledge, while the `cumulative` program is meant to track the knowledge that should be accumulated over the time. Lastly, the `volatile` program contains the rules that are time dependent but transient after a specific number of steps. It keeps the history accumulated through a specific range of time. Program 2.1.3 demonstrates the structure of an incremental program. The program may have more than one volatile parts with different volatile window durations. The external predicates that are being fed to the system through the *controller* program, are indicated with "`#external`" label. The incremental grounder grounds the time dependent parts of the program as the time parameter increases. Assuming that these three parts are indicated respectively by B, P, and Q, and the window for the volatile part is of the size j, the task of the reactive answer set solver at time point k would be to find answer sets for the program

$B \cup_{1 \leq i \leq k} P[t/j] \cup_{k-j \leq l \leq k} Q[t/l]$ given the online input data from time steps up to the time point k. Their approach in dealing with a dynamic parameter inside the ASP solver provides the means for handling the history of the domain inside the ASP solver itself.

The Reactive Answer Set Solver, *oclingo*

In this section we further describe how *oclingo* implements the idea of reactive answer set programming. In summary, *oclingo* takes an incremental logic program, listens on a port to get online data from an external source, augments the arrived data into the incremental logic program, and solves the program for each time step. It synchronizes the external online input time steps with the underlying incremental programs steps so that the resulting answer set(s) from each time step only take into account the online external input up to that point in the time. The input language for *oclingo* is the same as *clingo* except for some changes in the format of the program. *Oclingo* takes an incremental logic program as input and acts on it given the user external online input that is fed to the system through an external controller. The online input gets introduced into the logic program as external predicates. A separate controller program is responsible to pass the synchronized online input to the solver for each time step. The online input for each step must be presented in the following format:

```
#step k. external_predicate. #endstep.
```

The overall process of the solver is as follows. The solver should be launched with an incremental program and an online input data file. The first step for the solver is to ground the *base* part of the program and calculate the answer set(s) without taking into account the external input data. The solver then goes through an iterative process of taking the external online input from each step and calculating the answer set(s) given the input. At each iteration the solver performs the following tasks:

- ground the external input from step i and add it to the solver[12];
- ground the cumulative part $P[t/k]$ and add it to the solver;
- ground the volatile part $Q[t/k]$ and add it to the solver; remove the expired ground rules $Q[t/j]$ where $j < k$ is the biggest index which is outside of the range of the valid

[12]The current implementation limits the external input to ground atoms, so the input is already grounded

window[13].

- solve the program given the added ground rules.

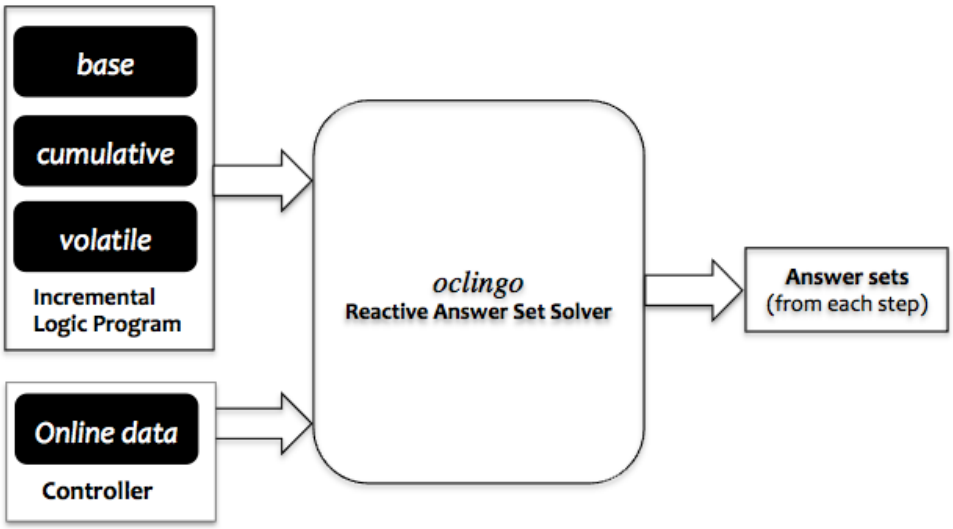

Figure 2.2: *oclingo*, Reactive Answer Set Solver

Gebser et al. [11] further develops *oclingo* by introducing the idea of a sliding window and improves the flexibility of defining problems by enabling arbitrary use of time-decaying rules and data, in both the encoding and the external input data. To this end, they extend the input language of *oclingo* to better deal with expiring data.

2.1.4 Problem Solving using ASP

This section describes the general methodology for encoding problems using ASP programs. As we have mentioned before, in declarative programming languages the programmer is mostly concerned with *what* is to be computed, rather than *how* to compute it. Therefore an ASP program describes the problem as opposed to presenting an algorithm to solve the problem. An ASP programmer needs to encode the problem in a way that the program itself is a precise description of the problem, and the resulting answer set(s) correspond to valid solution(s) for the problem. The general approach in coding a problem in ASP is the

[13]The process of removing expired rules is performed by turning the invalid rules into integrity constraints and adding them into the solver

"*generate* and *test*" [19] method. The *generate* phase uses choice rules to generate a set of potential solutions which is actually a superset of valid solutions. The *test* phase will then eliminate the solutions that are not in accordance with domain constraints. For example, when encoding an ASP program to schedule marine traffic through canals, we can start by generating all the potential solutions where every vessel can go through all the lanes.

$$\{\texttt{start\_request}(\texttt{I},\texttt{L},\texttt{T}) : \texttt{time}(\texttt{T}) : \texttt{lane}(\texttt{L}) : \texttt{vessel}(\texttt{I})\}. \tag{2.11}$$

where `I` stands for vessel id, `L` indicates the lane id, and `T` is time step. In the next step, we need to eliminate invalid answer sets by imposing the constraints. In the above example, one of the constraints is not to schedule a vessel on two different lanes at the same time.

$$:- \ \texttt{start\_request}(\texttt{I},\texttt{L1},\texttt{T}),\ \texttt{start\_request}(\texttt{I},\texttt{L2},\texttt{T}),\ \texttt{lane}(\texttt{L1};\texttt{L2}), \texttt{L1} != \texttt{L2}. \tag{2.12}$$

In summary, to represent a problem using ASP we first need to have a precise description of the domain and then go the through "*generate* - *test*" steps. The *if–then* form of ASP encoding makes it very similar to natural language and allows for predicates to represent real world concepts. This makes the code easy to read and intuitive.

2.2 Situation Awareness

Endsley [5] elaborates on the definition of situation awareness of "knowing what is going on around you" and advances it to more a practical definition as "the *perception* of the elements in the environment within a volume of time and space, the *comprehension* of their meaning and the *projection* of their status in the near future." According to this definition, in a situation awareness system

> "...at the lowest level the operator needs to *perceive* relevant information (in the environment, system, self, etc.), next *integrate* the data in conjunction with task goals, and, at its highest level, *predict* future events and system states based on this understanding."

There are three phases in this situation awareness model:

- **Perception**: The perception phase deals with low level information from a variety of sources.

- **Comprehension**: The comprehension phase comprehends the information in conjunction with background knowledge to understand the world.
- **Projection**: The projection phase predicts possible evolutions of the state. A prediction is considered to be one or more guesses on how an ongoing scenario may turn out.

In this thesis we have adopted the State Transition Data Fusion (STDF) model [17] for situation awareness (SA) which elaborates the above definition in a data fusion context. STDF has a systematic approach to manage the complexity of SA systems through modularization.

The situation awareness models [1, 17] based on this definition perform analysis in three levels of abstraction. The first layer is called *object assessment* and it is responsible for structuring the input data and extract object representations from the raw input. The second layer, *situation assessment*, discovers the relationships between objects and generates *situations*. Finally the *impact assessment* layer tries to find out the effects of situations and the relationship between situations distributed both in time and space. Each of these layers contain three components corresponding to *comprehension*, *projection*, and *prediction* phases in the definition by Endsley.

In order to make the system design more systematic, an abstract state machine (ASM) approach is combined with the STDF model. ASM's enable a stepwise refinement of the model from a highly abstract model to a coherent detailed implementation throughout the design process. We have used the *CoreASM* modelling framework[6] to apply the ASM approach. *CoreASM* has been used to implement the high-level SA model as well as the first two layers of analysis; while the tasks from the third layer are delegated to an ASP system. In the rest of this section we will be looking at the STDF model and *CoreASM* in more detail.

2.2.1 The STDF Model

The State Transition Data Fusion (STDF) model expands upon Endley's definition of situation awareness model [17] by using a state-based data fusion model. *Data fusion* is generally defined as:

> "...the process of utilizing one or more data sources over time to assemble a representation of aspects of interest in an environment" [16].

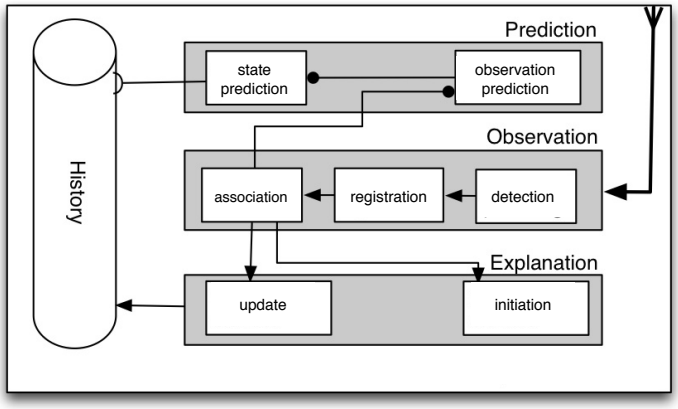

Figure 2.3: The STDF Model [17]

The STDF model represents a system in terms of states and transitions between the states, where the world is denoted as a set of states at each time step. STDF achieves situational awareness by assessing the input in three levels of abstraction:

- *object* assessment:
  raw input is transformed into objects in the world model.
- *situation* assessment:
  relations between objets are represented.
- *impact* assessment:
  effects of relations between objects are identified.

The internal structure of each of these layers (object, situation, impact) follows the SA definition by Endsley. They each contain an *observation*, an *explanation*, and a *prediction* component respectively corresponding to *perception*, *comprehension*, and *projection* phases from the SA definition.

As an illustration, when applying the STDF model to our Maritime Traffic Control problem; the *object* assessment layer structures the raw input by detecting objects and assigning them their properties such as speed, country of origin, location, etc. The *situation* assessment layer recognizes the relations between objects as the ongoing situations in the environment; we can then pick the situations that are of interest to our application. For

example, determining that two tanker vessels with one carrying ammonia and the other one carrying bleach are in a close proximity is called a *bad cargo coincidence* and is a situation of interest in our application. Lastly, the *impact* assessment layer identifies effects of interacting situations from different time points. As an example, looking at two situations from different time points where in each of them there have been two tankers from a specific country involved in a *bad cargo coincidence*; one may consider it as an anomalous action. Each of the layers pass their output as input to the next layer in order to perform more abstract analysis.

As we move from object assessment to situation assessment to impact assessment, the level of abstraction goes higher and with it the observational, explanatory and predictive components become more powerful. At each time step, the *observation* component in the *object assessment* layer deals with signal level sensory input in order to structure the input into *objects*; while the input of the *situation assessment* layer are the resulting objects from *abject assessment* layer and therefore it is more abstract compared to the initial sensory input. Similarly, the situational facts drawn in the *situation assessment* layer get passed as the input for the *impact assessment* layer analysis.

Figure 2.3 shows the overall STDF model, where each of the main components (Observation, Explanation, and Projection) are composed of more specific purpose modules that are also replicated through the three layers of analysis.

In this model, the world is understood in terms of a set of *states* and the *transitions* between them. At any time step k, the world is composed of a number of states. Depending on the level of analysis the state representation is of a different abstraction level; i.e. *objects*, *situations*, *impacts*.

There is a common pattern in the inner arrangement of the components of the *object*, *situation*, and *impact* assessment layers; however, the abstraction level of the layer changes the functionality of these components as the state representations vary at each layer. The overall task in the *object assessment* layer is to turn low level signal inputs into *objects*. For *object assessment*, **states** include *objects* with measurable properties represented. An object at time point k is represented as a vector $v_i(k)$ containing properties of the object i at the current time. However, the state of each object at time point k is represented as $s_{obj_i}(k) = \{v_i(t) \mid t \leq k\}$ and therefore includes the history of the object in the state. One can assume the overall world representation in object assessment process at time point k as a set of states of objects present in the domain $S_{obj}(k) = \{s_{obj_i}(t) \mid i \in objects \ \& \ t \leq k\}$.

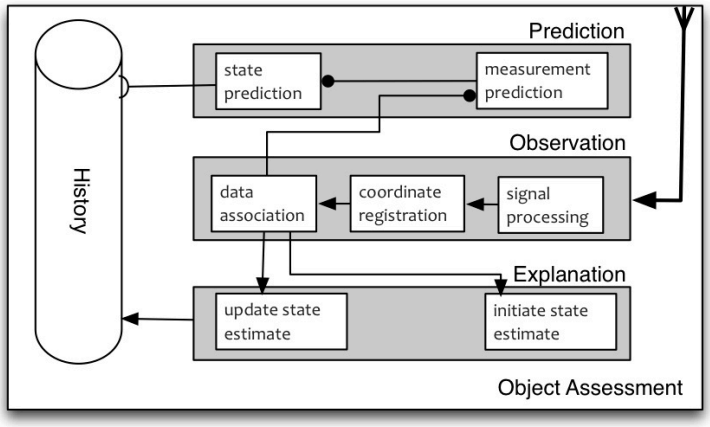

Figure 2.4: The STDF Model for Object Assessment [17]

Throughout this process, environmental sensors gather information and pass it to a *detection* module in order to identify the signals and distinguish false signals from the valid ones. A *registration* module then normalizes detections relative to a frame of reference. The *Association* process then invokes the prediction module to perform analysis given the history from the previous states. It matches the observation from the current state against the predictions from the previous steps to see if there is a match between current observations with predictions from previous observations; it then returns the analysis results back to the association process. The *association* process then moves to the *explanation* process to either update an existing state or to initiate a new state if there is no match between the current observation and some prediction from the previous state.

The other two layers, *situation assessment*, and *impact assessment* layers follow a similar process with a more abstract representation of the **states**. As shown in figure 2.5, the situation assessment layer locates the object assessment layer instead of the *detection* component and proceeds to the rest of the process using the resulting objects from the object assessment layer.

The impact assessment layer replaces the detection component with the object assessment layer and registration module with the situation assessment layer.

The representation used in each of these two layers are described in the rest of this section.

In *situation assessment*, the world is viewed in a more abstract fashion, in terms of *situations*. **States** consist of situations that are expressed as *set of set of statements* about

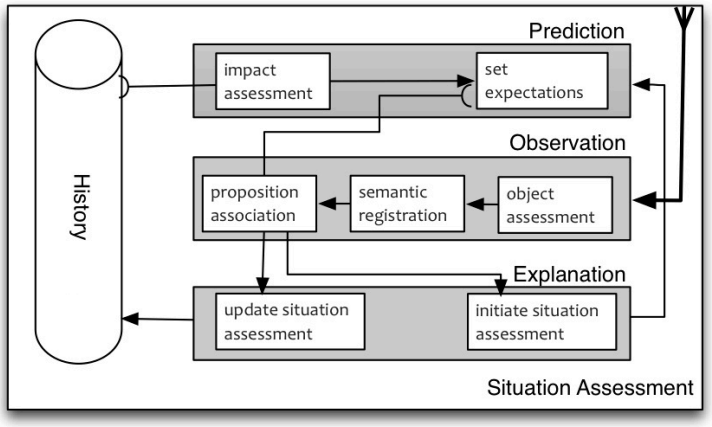

Figure 2.5: The STDF Model for Situation Assessment [17]

the world. A situation $sit_j(t)$ at time point t is a statement about the world explaining the relation between two or more objects at a time point. A situation can be represented as a statement in some formal language. For example, one may represent a situation involving two vessels, $V1$ and $V2$, in a *bad cargo coincidence* situation in a statement in the following form:

```
bad_cargo_coincidence(V1,V2,Area,t)
```

A situation may evolve over time and therefore its state is expressed as a set of statements. A state at time point k is represented as $s_{sit_j}(k) = \{sit_j(t) \mid t \leq k\}$; the state representation at time point k includes the history of how that situation has evolved over time. The set of states forming the entire world at time point k can then be represented as $S_{sit}(k) = \{s_{sit_j}(t) \mid j \in situations \ \& \ t \leq k\}$

For *impact assessment*, states are defined in an even higher abstraction level compared to *situation assessment* layer; **states** are defined in terms of *scenarios*. A scenario is expressed as *a set of interacting situations* which result in some effects that are of interest in the domain. A set of interacting situations, $interacting\_situations(l)$, is subset of situations in the domain that are somehow related to each other. A state in the impact assessment layer is a scenario which is defined as a set of interacting scenarios, $senario_l = \{s_{sit_m} \mid m \in interacting\_situations(l)\}$. The overall world in impact assessment layer is represented as a set of scenarios, $S_{imp}(k) = \{senario_l \mid l \in set\_of\_interacting\_situations\}$

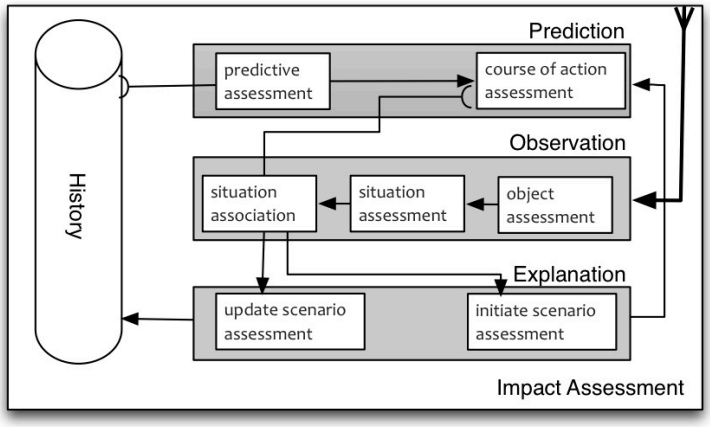

Figure 2.6: The STDF Model for Impact Assessment [17]

We have adopted the STDF model and customized it to perform the situation assessment task in our Marine Traffic Control problem. As explained before, STDF model takes the world in terms of states and transitions between those states; therefore we have used a state-based modelling framework, called *CoreASM*, to implement the situation awareness model. Section 2.2.2 discusses *CoreASM* and its underlying theory in some detail. Later on in Chapter 3 we explain the customized version of the STDF model and explain the implementation in further depth.

2.2.2 *CoreASM*: An ASM based Modelling Framework

Abstract State Machine (ASM) modelling is a method for design and analysis of complex systems. ASM provides a framework to model discrete dynamic systems by combining the concept of *abstract states* with *transition systems*. The main idea in the ASM modelling method is to enable a *pseudocode* like encoding to operate on *abstract* data structures where every aspect of the system can be modelled at any desired level of abstraction [8]. *CoreASM* is one of the implementations for ASM that provides a framework for rapid prototyping of abstract executable models[6]. It adopts its modelling method from ASM and provides a pseudocode like language with high-level abstract data structures. A basic ASM model[3] is defined as follows:

Definition 1. *A basic ASM M is defined as a tuple of the form (V, I, R, P_M) where the vocabulary V consists of a finite set of function, I is a set of initial states for V, R is a set*

Program 2.2: *CoreASM* Specification – Sorting

```
while (|set| > 0)
choose x in set with (forall y in set holds x <= y) do
    remove x from set
    add x to sorted
```

of transition rule declarations, and $P_M \in R$ is a distinguished rule, called the main rule or the Program of machine M.

Each function name f has an arity which is a non-negative integer. Function names with arity zero correspond to the concept variables in most of the imperative programming languages. An interpretation $f : X^n \rightarrow X$ for each function is provided at a given state S; i.e. a value v is assigned to each *location*, $f(v_1, ..., v_n)$. The set of all $(f, (v_1, ..., v_n), v)$ that hold in a given step form a state. The evaluation of a transition rule in a given state produces a finite set of updates of the form $(f, (v_1, ..., v_n), v)$. An update $(f, (v_1, ..., v_n), v)$ prescribes a change to the content of location $f(v_1, ..., v_n)$ taking effect in the next state.

ASM models are pseudocode programs allowing arbitrarily abstract operations on arbitrarily abstract data structures. The *CoreASM* specification language and its underlying semantics are claimed to be identical to those of ASM. A *CoreASM* specification is a pseudocode program that starts the execution from a main program called *init* and enables highly abstract operations using *transition rules* on abstract data structures. Program 2.2 displays a sorting algorithm implemented using *CoreASM*. As shown in Program 2.2, the programmer does not need to be involved in the fine details of each step of the algorithm. Gabriele [10] provides an elaborated example simulating a *Train Control System* using the ASM and *CoreASM*. See [10] for a detailed description and the *CoreASM* specification.

Stepwise refinement is the key modelling clue in ASM framework[3] that helps manage the complexity of the system by implementing the system at suitable levels of abstraction and linking them down to a concrete model. Using an ASM framework one can (*i*) form a base model given the high-level requirements in a precise yet abstract fashion, (*ii*) incrementally refine the base model down to a concrete implementation, (*iii*) validate the model through simulation or testing at each level of abstraction.

As described in the above steps, the main design clue is to start with a highly abstract, yet executable, model and go through a refinement cycle to improve and sketch the model

down to the desired concrete model.

Chapter 3

Situation Awareness in the *Marine Traffic Domain*

Maritime traffic control is an example of a task that looks for methods to enhance the capabilities of human operators as much as possible. Maritime operators, who watch over the oceans 24/7, are responsible for ensuring vessels comply with maritime regulations. In order to do this, the operator must first analyze large amounts of data from various sources ranging from satellite images to guard reports. The operators must use their expertise to combine the relevant data and infer additional situational facts in order to determine if further action is required. However, with the vast amount of information along with the numerous rules for safety compliance, we can see how the problem quickly becomes intractable for a human operator without some automated assistance to evaluate the large number of situational facts that can be derived.

A *situation awareness system* can be utilized to augment the human operators' expertise and analyze the available information to detect anomalous[1] actions and events. As an illustration, if a ship is moving faster than its allowed speed, the system can consider it as being anomalous. Unfortunately, unlike the speed limit checking, not all of the anomalous activities are easily observable by processing the low-level information; some situational facts are not directly observable and need expertise and more abstract information to be inferred. Therefore, the situation assessment model needs to perform the analysis in different levels of abstraction from low-level input data analysis to higher level information analysis.

[1]The term "anomaly" in this domain can be defined as deviation from the expected [26], such as those that would be related to smuggling, piracy, or potential hazardous conditions

One of the other challenges in the maritime domain is its highly dynamic nature. A constantly changing environment requires us to maintain a history of objects so we can detect anomalies where information from previous time points are required.

A situation aware system would be able to draw appropriate conclusions based on the existing information and to act according to its understanding of the environment. In order to implement a system able to act upon its knowledge and information from the domain, we have adopted the State Transition Data Fusion (STDF) model for situation awareness. The customized STDF model enables performing each of the anomaly detection tasks in an appropriate level of abstraction. The modular nature of the STDF model allow us to utilize appropriate tools to address each task.

As mentioned in the previous chapter, ASM facilitates the creation process by letting the system design start from a highly abstract model and be structured to the desirable detailed model through stepwise refinement of the model. It allows the designer to focus on the essential aspects of the system rather than encoding insignificant details. The *CoreASM* modelling framework follows the design approach proposed in ASM and therefore allows the designer to implement abstract executable specifications of the system at an arbitrary level of abstraction. It enables the designer to incrementally go from a highly abstract model to a concrete and fully detailed specification. We use the *CoreASM* modelling framework to implement the customized STDF model. *CoreASM* provides a multi-agent framework allowing parallel and sequential execution of the instructions.

The *CoreASM* implementation is responsible to delegate each of the specific anomaly detection tasks to the appropriate module to be addressed. Depending on the nature of the tasks, they are performed either in an imperative program using a *CoreASM* specification, or through an ASP program in a declarative fashion. The *object* assessment analysis dealing with low level input data as well as *situational* assessment tasks that follow simple algorithms are performed through a *CoreASM* specification in an imperative fashion. ASP is used for the tasks that involve higher level rules dealing with more abstract information in the *impact* assessment layer. As described in section 2.1.3, an incremental and reactive implementation of ASP, *oclingo*, is used to manage dynamicity of the domain inside the system. This implementation allows us to keep track of the domain history in the ASP program itself, as opposed to previous implementations where the history would need to be handled through an external component.

This chapter continues with a description of the part of the maritime domain which is

addressed in this work. Section 3.2 will then present the customized version the STDF model used for modelling situation awareness, as well as an explanation on how the *CoreASM* modelling framework is applied. The ASP component of the model is discussed in the following chapter.

3.1 *Marine Traffic* Domain Description

The maritime domain as a whole is a huge domain as it involves so many entities with a vast number of situations to be aware of and tasks to be performed. There are many types of anomalies that the marine traffic operators need to detect. Smuggling, piracy, dangerous cargo coincidences, and vessels in restricted areas are just a few examples of the anomalous situations that marine operators need to be vigilant for. On the other hand, there are also tasks that are not necessarily anomalous but still need operator consideration. For example, in harbour where the cargo vessels and tankers need to unload, there should be an agent scheduling the resource. In this work we design a situation awareness model to help in detecting the following anomalous situations in their appropriate level of analysis, as well as a harbour scheduling task:

1. vessels in prohibited areas - addressed in *Object assessment* layer
2. dangerous cargo coincidence - addressed in *Situation assessment* layer
3. suspicious vessel coincidence - addressed in *Impact assessment* layer
4. marine resource scheduling - triggered by the *Object assessment* layer

The first three situations listed above are chosen so we can describe all the abstraction levels of analysis in our situation awareness model; i.e. object assessment, situation assessment, and impact assessment respectively. The last scenario was chosen in order to show how a declarative paradigm can be useful for a general search problem in a dynamic and rule-based domain. One of the challenging tasks for marine traffic officers is directing vessels in areas of high density and a narrow lane to prevent dangerous maritime situations [21] [9]. Harbour unloading platforms is another example of a marine resource that needs to be organized using a scheduler. We address the marine resource scheduling problem using an ASP module.

The rest of this section is devoted to a more detailed explanation of the marine traffic control domain. Since the entire maritime domain is rather expansive, we took a small portion as a sample so that we can illustrate the suitability of our model together with ASP and *CoreASM*.

The following gives a rough idea of what the maritime traffic domain entails:

> Vessels travel along shipping lanes going from one terminal to another. These vessels comprise of passenger ships, cargo ships, and oil tankers. Each of these vessels have various priorities for passing through channels or for docking at harbours to unload cargo. In order to gather information about the various vessels in the area, an Automatic Identification System[2] (AIS) is used. This system tracks vessel positions, speed, etc. The main role of the marine traffic officers is to use the information gathered by the AIS to make a determination of the current state of affairs. Having an accurate sense of the current situation is required in order to make informed decisions and to properly react to suspicious or unusual activity. The marine traffic operators are also responsible for scheduling the vessels by taking their priority into consideration while at the same time ensuring that the proper safety regulations are being adhered to. Detecting vessels in prohibited areas, recognizing dangerous cargo coincidences, and discovering vessels with suspicious behaviour are some of the tasks that we will be looking at. A dangerous cargo coincidence happens when two ships carrying dangerous chemicals are not at a safe distance apart from each other.

We perceive the domain as being composed of two main agents: the *environment*, and the *base station*. The environment is where all the objects operate and actions are performed. The AIS system passes information obtained from the environment to the base station. The AIS again is an automatic tracking system that provides unique identification, position, course, speed, etc. of the objects in areas of interest. The base station is the agent that receives all the data and information from the environment. It analyzes the information and tries to maintain a situation-aware state given the observations. In order to enable the base station to achieve situation awareness, the SA model is located inside the base station. We simplify the domain by assuming that the vessels are the only dynamic objects present in the environment. **Vessels** are moving objects that are observed in the environment through

[2]We have used an AIS available online at http://www.marinetraffic.com/ais/

the AIS. As discussed earlier in section 2.2.1, the STDF model represents domain objects as vectors containing properties.

$$v_i < position, speed, course, area, type, time >$$

where *position* of the vessel is composed of a latitude and a longitude. The *type* of the vessels determine their priority in using resources; for example, cargo vessels carrying dangerous goods may need safety considerations and have a priority over other vessels.

Low level analysis on the input observation needs to be performed in order to identify each vessel and its status in the domain. An estimation of the vessels' future location given the current latitude, longitude, course, and speed is one of the properties of interest in this domain. Compliance of the vessel with basic rules in the domain is another factor that identifies the status of the vessel in the domain. For example, one needs to ensure that vessels do not exceed the speed limit. Another example would be checking the vessel location and ensuring that it is not in a prohibited area. After clarifying the status of each vessel, the interaction of the vessels with each other needs to be analyzed. Determining if there are any vessels carrying conflicting cargos in a close proximity is called a *dangerous cargo coincidence* and is a situation of interest in our application. This pattern of interaction of the vessels can also help us detect suspicious vessel coincidences. Vessels involved in a set of situations are called suspicious if the situations are considered to be related to each other according to the expert knowledge.

The rest of this chapter fits the marine domain described above into a situation awareness model.

3.2 Situation Awareness in *Marine Traffic* Domain

In this thesis we have adopted the State Transition Data Fusion (STDF) model [17] for situation awareness (SA). The STDF model offers a systematic approach to manage complexity of SA systems through modularization. Pairing the STDF model with the Abstract State Machines (ASM) approach enables stepwise refinement of the model from a highly abstract model down to a detailed implementation throughout out the design procedure.

We have used the *CoreASM* modelling framework[6] to apply the ASM approach. Combining the STDF with the ASM method provides a systematic modelling approach to manage complexity of the system through modularization and refinement. *CoreASM* has been used

Program 3.1: *CoreASM* Specification – STDF Program

```
rule STDFProgram(time) =
seqblock
    ObjectAssessment(time)
    SituationAssessment(time)
    ImpactAssessment(time)
endseqblock
```

to implement the high-level SA model as well as the first two layers of analysis; while the tasks from the third layer are delegated to an ASP system. In the rest of this section we will be looking at the STDF model and *CoreASM* in further detail.

As mentioned earlier, we divide the world into two agents, environment and the base station. The *environment* agent simulates the environment based on data provided by an AIS. The second agent is the *base station* which performs the SA analysis. These two agents are represented as agents in the *CoreASM* specification.

The base station is where the situation awareness model is located. The *CoreASM* specification, presented in program 3.1, displays the overall STDF model for situation awareness. The STDF model performs the situation analysis by understanding the world in terms of objects, interaction between them, and scenarios composed of events occurring in the domain. It predicts the possible events and evolutions of the world at each level of abstraction. Each of the *object*, *situation*, and *impact* assessment layers observe the world, and explain it in terms of objects, situations, and scenarios respectively. Each layer then makes predictions on how it would evolve in the near future. The analysis from the first two layers, *object* and *situation* assessment, deals with numeric features of the domain and therefore opts for an imperative language to perform computations. However, the *impact* assessment layer deals with higher level statement and contains no numeric information; therefore we have used a declarative language for the analysis in the latter layer.

Figure 3.1 depicts the main components of the SA system and their interaction. As shown in the diagram, the *CoreASM* component performs the analysis from the object assessment and the situation assessment layers; while it delegates the impact assessment analysis to the ASP component where the basic situational facts are supplied from the previous two layers. The ASP component is composed of an *incremental logic program*, where the expert knowledge is encoded, a *controller* program, that passes the external situational facts to the engine, and an ASP-solver engine performing the inference task.

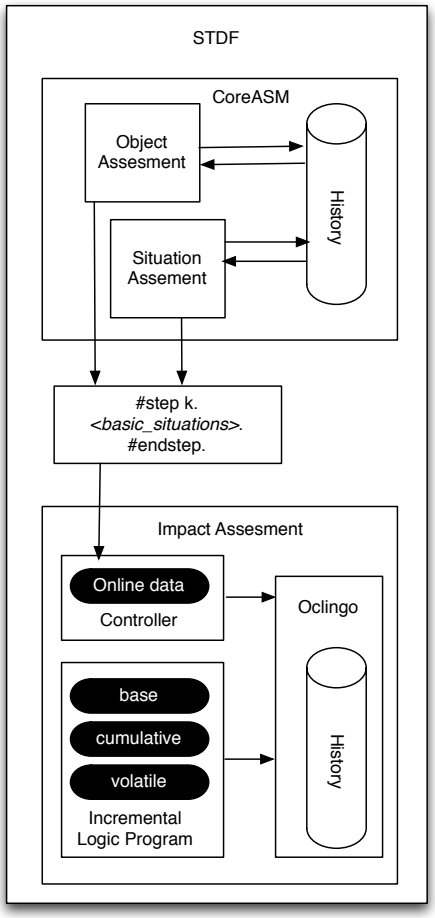

Figure 3.1: Situation Awareness Model Using *CoreASM* and ASP

In the first layer of analysis, *object assessment*, we are concerned with gathering all the basic and measurable properties, as well as the basic situational facts regarding the status of each vessel as an object in the domain. The *object assessment* process produces an object representation report and draws all the inferable object level facts. It performs the analysis through three components: *object observation*, *object prediction*, and *object explanation*. Program 3.2 displays the *CoreASM* specification implementing the *object assessment* layer by delegating the task to the *observation*, *prediction*, and *explanation* modules respectively. As described in section 2.2.1, the *object observation* module divides the task into three sub-components: *detection*, *registration*, and *association*. The *detection* module is responsible for identifying the signals and distinguishing false signals from the valid ones. In this

Program 3.2: *CoreASM* Specification – Object Assessment

```
rule ObjectAssessment(time) =
seqblock
    mode := OObservation
    ObjectObservation(time)

    mode := OPrediction
    ObjectPrediction(time)

    mode := OExplanation
    ObjectExplanation(time)
endseqblock
```

application, the input information is not presented as low level signals as the AIS performs the low level signal processing and provides our system with data such as latitude, longitude, speed, etc. The *registration* module structures the detections and compiles object vectors with their basic properties provided by the AIS.

$$v_i < latitude, longitude, speed, course, type >$$

The *association* module interacts with *object prediction* and *object explanation* components in order to complete the object representations. It first takes the basic object representation from the *object observation* component to the *object prediction* component in order to augment the state representation with predictions about the observations. It accesses the history of object representations in order to decide if the object is newly added to the domain or has already existed in the previous steps. In order to simplify the process and avoid redundancy, we ignore the *association* module and distribute its tasks into other components. We allow the *registration* component to access the history and be able to take care of adding new states for the newly added objects and updating states for the existing objects. The *object prediction* component clarifies object states by augmenting the state vector with additional information. An example of such additional information is predicting object positions in the next two time steps. The *object explanation* component performs on the object vectors and draws basic facts regarding the status of each object in the domain. Given the object vector, we can use all the basic properties of the object, e.g. its current location and its predicted future locations, to draw some basic facts such as:

- if the vessel is in a prohibited area.

Program 3.3: *CoreASM* Specification – Detecting Vessels in Prohibited Areas

```
rule DetectVesselsInProhibitedAreas(time)=
    forall v in Vessels do
        if <conditions hold> then in_prohibited_area(v,area,time):=true
```

- if the vessel is heading towards a prohibited area.

These situational facts are represented as first order propositions in the following form:

$$situation(< parameter\_list >)$$

Each situation in the domain corresponds to an event in the environment. For example, the situation where there is a vessel in a prohibited area we get:

$$in\_prohibited\_area(vessel, area, time)$$

Such situations are inferred through the analysis performed at each module. For example, the rule displayed in Program 3.3 determines if there are any vessels in any of the designated areas. As mentioned earlier, the situational facts inferred in the *object layer* only identify the status of each object in the environment independent of its interactions with the rest of the objects in domain. Another example of a proposition explaining the status of a vessel in the domain is the state of arriving at a harbour to unload the cargo. A request to unload cargo in a harbour can be represented in the following format:

$$resource\_request(vessel, harbour, duration, submission\_time, priority)$$

This request is then passed to the resource scheduler to organize the unloading queue. The scheduler module in this application is implemented using ASP. The next chapter will provide an explanation of how we can benefits from addressing this problem with a declarative language.

The *situation assessment* layer takes the resulting object representations as well as the basic situational statements and discovers the relations between the objects in the domain. This layer accesses the numerically based features of objects as well as propositions explaining status of objects in the domain and draws higher level situational facts identifying the relations between the objects. The resulting outcomes from this layer are not numeric, they are rather expressing the interactions in terms of statements. The *situation observation*

component accesses the object states and the basic situational and passes them to the *situation prediction* to perform analysis on how the interaction of objects may evolve over time and set expectations of what are the possible future events. The *situation explanation* would then perform the analysis to explain the environment in terms of situational facts. These situational facts represent the interactions of objects in the domain. Resulting statements are first order propositions corresponding to events involving two or more objects in the domain. An example of such situations is a situation in which two vessels are involved in a dangerous cargo coincidence:

$$dangerous\_cargo\_coincidence(vessel1, vessel2, location, time)$$

The *impact assessment* layer is built on the previous two layers to perform higher level analysis based on the inferred situational facts about the objects and their relations. The impact assessment analysis represents the world in terms of scenarios that may involve multiple situations distributed in time and space. An impact assessment component should be capable of interpreting the situations, discovering the interacting situations, and predicting how a situation may evolve over the time. In practice, domain experts use their expertise to discover impacts in the domain. Due to rule-based nature of this process, we have opted for a declarative language to represent the rules used by the domain experts. Further details on how this component performs the impact assessment analysis is provided in the following chapter.

Chapter 4

Using ASP in a Rule-based Dynamic Domain

As described in the previous chapter and presented in figure 3.1, the *situation awareness system* takes the available information and passes that on to an *object assessment* layer where the input data is turned into objects with their properties, and the basic situations depicting interactions of objects with the domain are inferred. The object representations are then fed into a *situation assessment* layer in which the relationship between objects in terms of events involving more than one objects are drawn. The inferred situations from the previous two layers form the input for the *impact assessment* layer analysis. The *impact assessment* layer is where the ASP system analyzes the history of situations, discovers the relationships between them, and possible undesired interactions of events. In this application we will be looking for *suspicious vessel coincidences* where a vessel is identified as being suspicious given the history of the situations in which it has been involved. In this section we will be using ASP as a component in our three-layered *CoreASM* model; furthermore, we explore the use of ASP for a general scheduling problem in the domain.

The ASP module in this work has the responsibility of reasoning about the collected high-level situational facts and inferring additional situational facts. It does so by using reasoning rules which are expressed as logic program rules. Independent of the context of the domain, rules can be classified into two categories, (*i*) rules explaining domain expert knowledge and (*ii*) rules explaining the environment itself. In addition to the high-level situation awareness analysis, the ASP module has also been used to address the problem of

scheduling marine resources.

As a declarative language, ASP supplies rich means for representing rule-based domains. The non-monotonic nature of ASP rules makes it suitable for such domains as it provides a compact and intuitive encoding of the domain expert's knowledge. Moreover, real world rule-based domains are usually prone to policy (rule) changes which ASP seems to be quite flexible for such changes. Recent advancements to implementations of ASP enable incremental evaluation of logic programs; this allows for the program to have a notion of history which is essential for a *situational awareness* system.

The overall goal of this section is to demonstrate the suitability of ASP as a non-monotonic declarative language with an efficient reactive inference engine for the emerging tasks in a typical rule-based domain. The rest of this section will highlight use of ASP in discovering relations between situations and perform *impact assessment* analysis. The ASP system is also responsible for scheduling the resources given the awareness in terms of situational facts from the object assessment layer.

The remainder of this section is organized as follows: in section 4.1, we will see how ASP's expressive power becomes useful in encoding different aspects of the domain. Section 4.2 will then show how reactive ASP is used to analyze high-level information and detect undesirable situations. Lastly, in section 4.3, we present a compact module to address a general scheduling problem occurring in the domain in order to help marine traffic officers organize vessel traffic.

4.1 ASP Representational Potentials

In this section, we will be looking into representational aspects of ASP. As discussed in Chapter 2, as a non-monotonic logic, ASP has a rich representational potential which builds upon its expressive first-order like language. Also stated in Chapter 2, naf-literals provide ASP with a concise way of representing default rules as well as qualitative preferences. In addition to the language-independent representational potentials brought through negation-as-failure, *oclingo* has defined language constructs that enhance ASP's capability in dealing with history of the domain. In the rest of this section we will demonstrate how ASP provides a powerful means to express the marine traffic control domain.

Defaults

One of the important requirements when dealing with dynamic domains is existence of a mechanism to represent non-changing properties. This problem of non-changing properties is known as the frame problem. Negation-as-failure provides ASP with a concise solution to the *frame* problem[20] as it enables encoding of the rules in a way that allows a conclusion to hold in absence of additional information to the contrary. The following is an example of a rule assuming that the location of a vessel does not change from the previous time step to the current time step by default unless otherwise stated through new incoming information.

```
vessel_location(X,L1,t) :-
     vessel_location(X,L1,t-1),
     not ext_vessel_location(X,L2,t) : L1!=L2.
```

This concise encoding allows us to handle non-changing properties without having to list all the conditions to let a property hold.

Qualitative Preferences

Preferences can be assessed quantitatively by assigning numerical values to determine their preference, or they can be represented qualitatively as an implicit or explicit ordering between choices. ASP provides an elegant way of representing preferences implicitly through *Negation-as-failure*. It allow us to distinguish the default solution from a solution which is preferable in a specific situation. For instance, let's suppose that maritime protocols require the base station to send a notification to the vessel if they observe that vessel in a prohibited area. However, there is an overriding rule in more specific cases in which it is desirable to send the guard to the area if the vessel is behaving suspiciously. This scenario is an example of a situation in which we prefer an action in most cases but there is a more preferred action for a specific case. The following encoding demonstrates the above scenario in terms of ASP rules:

```
send_notification(A) :-
    vessel_in_restricted_area(V,A,t), not suspicious(V).


send_guard_boat(V,t):-
    vessel_in_restricted_area(V,A,t), suspicious(V).
```

Cardinality Rules

Cardinality rules let us define rules that require a specific number of literals to hold in order to allow the head atom be inferred. This type of rules appear in the following form:

$$head \ :- \ l\{a_1, ..., a_m\}u.$$

The body of a cardinality rule is a count aggregate and *head* atom holds if at least l and at most u number of literals in the set $a_1, ..., a_m$ are satisfied. This enables us to define rules where a specific number of conditions need to hold in order to draw some conclusion as opposed to requiring an exact set of literals to hold. For example, in the following scenario, we let the threat situation be resolved if at least one of the involved vessels is from a legitimate address.

```
threat_resolved(X,Y,t):-
     threat(X,Y,t),
     1{legitimate_addressee(X,C,t),
       legitimate_addressee(Y,C,t)}.
```

Weight Rules

Weight rules are a specific form of cardinality rules where a sum aggregate is used instead of a count aggregate. In the body of the rule, there is a weight assigned to each of the literals in the set and the sum of the weights of those literals in the set that hold are assessed. Weight rules are in the form of:

$$head \ :- \ l[a_1 = w_1, ..., a_m = w_m]u.$$

The *head* atom holds if the sum of the weight of the satisfied literals in the set add up to least l and at most u. This allow us to define a simple utility function and enables us define rules where a specific number of conditions need to hold in order to draw some conclusion as opposed to requiring an exact set of literals to hold.

Let's assume an example in which a threat will be considered resolved if at least one of the vessels had a legitimate addressee or they had both been inspected recently. In this case we can give a weight of two to having a legitimate addressee and a weight of one to vessels being recently inspected. We can then require it to hold a number of conditions that add up to at least 2 in order to resolve the threat.

```
threat_resolved(X,Y,t):-
     threat(X,Y,t),
     2[legitimate_vessel_addressee(V1;V2) = 2,
       vessel_inspected(V1;V2,T):time(T):T<=t = 1].
```

Transitive Closure

A transitive relation R is a binary relation where one can conclude for every a, b, c if aRb and bRc then aRc. Even though first order logic is one of the most expressive logical languages, it is unable to represent transitive closure. However, ASP offers a succinct encoding to address transitive closure. As an example, let's consider the notion of *later point in the time*. Assuming that time point t_1 and t_2 are successive time points where $t_1 < t_2$, we can conclude that t_2 occurs later than t_1. Let's consider time point t_3 which follows time point t_2. We can, thus, conclude that t_3 is later than t_1. This can be concisely represented by the following rules:

```
later(T2,T1) :- successor(T1,T2).
later(T3,T1) :- later(T2,T1), successor(T2,T3).
```

4.2 Impact Assessment

As discussed in the previous section, ASP provides a rich potential for representing rule-based domains. In this section we will see how ASP's expressive power combined with the facilities to handle history, offered by *oclingo*, provides a useful setup to perform impact assessment analysis in a rule-based dynamic domain. The suspicious coincidences of vessels is an example of a situation where the marine officer would consider a set of situations as being anomalous, if they observe some specific patterns relating those situations to each other.

As described earlier in section 2.2.1, we define a scenario as a set of interacting situations which results in some effects of interest (impact) in the domain. Let's consider a scenario in which a vessel would be considered as being suspicious if the following pattern is observed:

> *... two vessels have been involved in more than one dangerous cargo coincidence situations together in the past two months and at least one of them has not been moving outside of the area for more than a week.*

In this scenario the case would be considered as being safe if:

> *... at least one of the vessels is from a legitimate addressee or the vessels both have been inspected in the last two days.*

The impact assessment component is implemented using the reactive answer set solver, *oclingo*. This component is responsible for analyzing the history of situations, figuring out the relations between them, and the possible suspicious interactions of events.

Propositional statements, representing situations involving one or multiple vessels, form the input for the ASP program. As described in section 3.2, the input for the impact assessment component is supplied through the output from the object and situation assessment layers. The object assessment layer infers the situational facts involving a single vessel; while, the situation assessment analyzes interactions involving multiple vessels. The resulting propositional statements from these two layers are then encoded in terms of *oclingo*'s external input language and is passed to the *controller* program. As described in Section 2.1.3, *oclingo* augments the *incremental logic program* with the *online data* at each incremental step and solves the resulting program. The incremental logic program represents the impact assessment scenario rules where the incoming situations, representing the current state of the world, need to be matched against the scenario rules in order to discover impacts of the courses of actions.

Depending on how complex the scenario is, it can be encoded using one or multiple ASP rules. The facilities provided in the *oclingo* input language, built upon the original ASP language, makes the encoding rather succinct.

In order to encode a scenario, the impacts of the scenario are mapped to head predicates of the rules; while the basic situations are mapped to body predicates representing the conditions where an impact can be inferred. The next step is to determine which part of the incremental logic program is the right place for the rule; *base*, *cumulative*, or *volatile*. The rules encoding the scenarios need to be assessed at each step in order to infer possible impact; therefore, there is a notion of time in the rules encoding the scenarios. Thus they should be placed in one of the time-dependent program parts; i.e. *cumulative*, or *volatile*. The *cumulative* part allows persistent accumulation of the history; while the *volatile* part can be used to keep track of the history for a specific duration. The persistent rules need to be located under the tag "`#cumulative t.`" indicating the cumulative program part; while the volatile rules are placed under "`#volatile t : duration.`" with an arbitrary duration.

The choice between these two program parts depends on how far we need to maintain the history of the inferred facts in the system. For example, one may assume that the history of the vessels that have been recognized as being suspicious should never be removed from the system and therefore be placed in the *cumulative* part. However, this may not be a good decision since the persistent atoms can occupy a large amount of memory in the long run, and result in regressive performance of the system. For the atoms that we only need to be present in the system for a specific period of time we can define a volatile part and place the rule in it. The incremental program may have several volatile parts with different durations; therefore making it flexible to deal with history at the rule level.

The history for the basic situations which are being fed to the system through the *controller* program can be handled likewise. For the input predicates we have the choice of handling the history in rule level and the input level.

For example, in the above scenario, the history of the basic situational fact indicating dangerous cargo coincidences needs to stay in the system for two months; however, the history of vessels being inspected would only be need to be kept valid for two days. Therefore, they can be defined to be volatile for a specific duration[1] of time.

```
#volatile : duration. ext_vessel_inspected(vessel1,t).
```

The resulting situational facts represent the impacts and possible events in the near future.

ASP's expressive power besides language-specific structures defined for *oclingo* provides a good setup for encoding complex conditions in a concise way. As we mentioned earlier, the rules can be considered as if-then statements where a predicate holds if its conditions are met; while the condition part in turn can involve complex criterions. Program 4.1 illustrates the encoding of the suspicious coincidence scenario, described earlier. In this scenario a predicate "`suspicious_coincidence`" is mapped to an impact in which two vessels are considered as being suspicious if the available basic situations match to the scenario pattern. In this case we would like the history of the suspicious vessels be persistent; therefore we defined them under the cumulative label in the program. Line 6-10 in program 4.1 encodes the condition: "*at least one of the vessels is from a legitimate addressee or the vessels both have been inspected recently*" by using a sum aggregate and assigning a weight of 2 to each vessel being from a legitimate addressee and assigning a weight of 1 to their being inspected in the

[1] a logical time step in this application is 90 seconds and thus two days would be 1920 logical time steps

Program 4.1: Suspicious Coincidence Scenario encoding

```
1  #cumulative t.
2  suspicious_coincidence(V1,V2,t) :-
3              dangerous_coincidence(V1,V2,T1),
4              dangerous_coincidence(V1,V2,t),
5              T1!=t,
6              not vessel_area_changed(V1;V2,T):time(T):T<=t,
7              [ legitimate_vessel_addressee(V1)=2,
8                legitimate_vessel_addressee(V2)=2,
9                inspected_vessel(V1,T):time(T):T<=t = 1,
10               inspected_vessel(V2,T):time(T):T<=t = 1]1.
```

last two days. As we explained in section 2.1.1, the sum aggregate returns the sum of the weights for the literals that hold. Setting the upper bound to one ensures that the aggregate statement would evaluate to false if the criteria for the scenario's being safe is met. In the other words, if the aggregate statement in the body evaluates to false, the case would not be considered as being a suspicious coincidence, as it means that either at least one of the vessels is from a legitimate addressee or they have both been inspected in the last two days.

As we can see in the above encoding, the criteria for the vessels being inspected in the "*last two days*" is not mentioned anywhere in the above rule. The reason is that the history of the external situational facts, "`inspected_vessel`(`V1`; `V2`, `T`)", has been handled in the input level by setting a volatile duration of two days. This means that the atoms "`inspected_vessel`(`V1`; `V2`, `T`)" would not be valid unless they occurred in the past two days.

In this example negation-as-failure is used to check the absence of information regarding the vessels' being outside of the area in the past week. Similar to the previous case, the history for atoms "`vessel_area_changed`(`V1`; `V2`, `T`)" have also been handled in the input level.

It is often the case that the policies in a rule-based domain can change or new scenarios are added to the system. The declarative form of the language has the benefit that adding a new rule corresponds to adding the description of the scenario to the program. In order to guarantee the proper functioning of the system when adding new rules we need to restrict use of *constraint rules* in the program. In a constraint-free[2] program, the resulting answer sets

[2]where there is no integrity constraint rules used in the program

would be identical to the combined answer sets when having a separate program for each set of rules. However, the current implementation of *oclingo* only supports incremental addition of ground (variable-free) rules and predicates; the general case of adding non-ground new rules to the program is under investigation by the *oclingo* development team.

4.3 Marine Resource Scheduling

In section 4.1, we illustrated ASP's rich potential in providing succinct representations for encoding different aspects of dynamic and rule-based domains. This section focuses on addressing the problem of scheduling marine resources which can be considered as a general search problem coming from a rule-based dynamic domain.

In order to address the problem we first need to provide a detailed description of the problem:

> There are a number of unloading platforms where vessels can submit requests to occupy them for a given duration of time. An unloading request contains: *vessel id*, the id of the requested *platform*, *duration* that it takes the vessel to complete the task, *submission time*, and the *priority* of the vessel based on its type.

A schedule for a set of tasks on a resource is an ordering in which we wish to let the tasks occupy the resource. The characteristics of an acceptable solution (schedule) to this problem is summarized below:

- do not allow conflicting tasks to be scheduled simultaneously; i.e. two tasks using the same resource at the same time.
- each task should only be scheduled once.
- an unloading vessel should not be interrupted by the scheduler before the completion of the task.

The vessel requests are gathered in the object assessment layer where the state of each vessel is identified. The situational fact representing the submitted request is represented in the following format and is passed to the scheduler program as an external predicate:

$$\texttt{ext\_request}(\texttt{vessel\_id}, \texttt{resource\_id}, \texttt{duration}, \texttt{submission\_time}, \texttt{priority})$$

The policy of the domain can specify the duration after which unscheduled tasks are expired; therefore, determining if the external predicate should be persistent or volatile. In order to keep track of waiting requests, a queue predicate is used; at each time step the incoming tasks as well as the ones from previous time step that could not perform their request are copied into the queue. Therefore, it helps with ensuring that each task is scheduled only once. Moreover, a predicate called `request_in_progress` is defined to indicate if a task is running.

As an illustration, lets consider the rule where we need to choose among waiting requests. This rule check two criterions for its choice; (i) the task is among the waiting requests, (ii) the resource is not being hold by another vessel. Following rule represents one of the possible encodings for this:

```
1   request_in_progress(I1,R,D1,T1,t) :-
2                       request_in_Q(I1,R,D1,T1,t),
3                       not request_in_progress(I2,R,D2,T2,t) : vessel_id(I2) :
4                       duration(D2) : time(T2) : I2!=I1.
```

where I indicates the vessel id, R is the resource id, D is the duration, T is the submission time, and t is the current time step. Line 2, ensures that only waiting tasks are considered to be valid. Lines 3-4, checks if the resource is not already occupied by another task. The domain expanders in the second part of the rule expand the variables in predicate `request_in_progress` over their domain.

Furthermore, we can easily specify additional properties on the solution. As an illustration, given that the policy in the environment changes and they decide to let vessels with higher priority be scheduled first, program can be modified so it adapts to the new policy and compute the desired schedule. In this example, we can simply specify modify the previous rule by augmenting the predicates with variable P indicating preference of requests and adding a criteria to make sure that the chosen request is not less preferred than any of the waiting request:

```
1      request_in_progress(I1,R,D1,T1,t,P1) :-
2                          request_in_Q(I1,R,D1,T1,t,P1),
3                          not request_in_progress(I2,R,D2,T2,t,P2) : vessel_id(I2) :
4                          duration(D2) : preference(P2) : time(T2) : I2!=I1,
5                          not request_in_Q(I3,R,D3,T3,t,P3): vessel_id(I3) :
```

```
6                          duration(D3) : time(T3): preference(P3): P1<P3.
```

Line 5-6 in the above rule ensure that the task is not less preferred than any of the remaining requests. We can also specify properties that take into account the situation awareness obtained in the system. For example, one may want to avoid scheduling vessels with conflicting cargos in subsequent time steps.

More interestingly, the language is flexible enough to allow switching between preference criterions to be applied on the solution. This is performed using a *qualitative preference* rule opting for either the tasks that have a higher preference number, or the ones taking a shorter duration to complete, depending on which is indicated as being more preferred.

The ASP encoding for the latter version of the scheduler that allows switching between preferred criterions is provided in Appendix A.1.

Languages allowing ordered disjunction enable better means for specifying qualitative preferences letting the answer sets be ordered based on the preference criteria. The optimization statements suggest yet another concise way of ordering answer sets when having multiple preference criterions; however, the optimization statements are not supported by the current implementation of *oclingo*.

Chapter 5

Discussion

In this section we will discuss the strengths and weaknesses of our approach. What we attempted to accomplish in this work was to analyze the capabilities of ASP to help solve the problem of situation awareness. The situational awareness task was abstracted into 3 levels; each of these levels were then further sub-divided into additional layers to deal with more specific tasks. ASP was chosen as the module to assist with the high-level analysis due to its powerful capabilities in representing abstract rules in a concise and meaningful representation. Furthermore, the recent reactive implementation of ASP offers a good set of means to handle the history of the rule-based dynamic domains in a seamless way.

Early implementations of ASP could only accept static input data; however, a recent reactive implementation of ASP enabled dynamic input data to be used and to generate answer sets incrementally. This reactive answer set solver provides a seamless way of handling the history of the domain inside the ASP system; it enables simple means to discard the information which is no longer useful. Previous implementations of ASP required iterative calls to the solver where no history from the previous runs were available in the subsequent iterations; however, *oclingo*, allows the system to run in an incremental fashion where a history of the information from the previous iterations is available. The flexible means to manage the history from the previous time-steps makes it rather useful for dynamic domains. As we illustrated in section 4.2, the history of the domain can be handled in both the input level and the program level where the former is more useful for the external situational facts, whereas the latter can be used to keep track of internally inferred situational facts.

ASP's intuitive encoding along with its rich representational potentials are the assets that make it popular for a wide range of applications in rule-based domains[2, 23]. We

have also demonstrated how representation of different aspects of the domain can benefit from the concise solutions provided by ASP; such as default rules, and qualitative preferences. Furthermore, we illustrated ASP's capability in expressing complex problems such as scheduling (refer to appendix A.1 for the encoding) in a fairly succinct encoding. This is due to the language being declarative and so the only parts that needs to be coded are the description of the problem and what the solution would look like. The steps to compute the solution are handled by the ASP solver even though the programmer may not know the algorithm to solve the problem.

ASP's declarative nature and the fact that the programmer does not need to encode an algorithm for every piece of information taken from the domain expert, makes it easier to migrate the expertise of the domain expert into the system. Furthermore, the resulting code is human readable, and therefore easier to confirm the correctness.

As we described in 4.2, we built an extensible setup for the impact assessment module implemented with ASP by restricting use of constraint rules in the program. This can guarantee removal of the interactions between the answer sets of newly added rules with the existing state of the program. Therefore, the resulting answer sets would be identical to the combined answer sets when having a separate program for each set of rules.

On the shortcomings side, there are two sets of practical issues with using ASP for situation awareness when applied in the real world. The first is the issue with accumulating history in a potentially long running time where the amount of history builds up in the system. When an incremental encoding involves many external atoms accumulating over time and consumes a considerable memory, the system may perform regressively.

The response time of our system, when provided with hand-input, is reasonably short; however, due to technical issues with the available implementation for the *CoreASM* framework we were unable to expose the system to real world input in order to see how the accumulating history effects the performance of the system. If the running time of the system were to become unacceptable with the accumulated history, a potential solution would be to use a sufficiently large volatile window so that we can keep track of long lived information, but still have it expire after some time. This would curtail the accumulation of persistent information at the expense of potentially losing relevant information if the volatile time window is not big enough. Although, this would help with avoiding the regressive performance of the system, it may not be a feasible solution for the applications where we need to maintain the history of events for an excessively long period of time.

Another performance related issue with using ASP when running on real world applications happens if there are a large number of atoms being generated in the initial grounding phase. If the predicates are defined on large domains it may result in an exponential blow up at the grounding process. However, abstract situational statements do not seem to involve a large domain. Therefore it would be unlikely for the high-level situational analysis with ASP to suffer from this issue.

The last practical hurdle for using ASP in a real world situation awareness system is the lack of a stable implementation which allows for dynamic input; *oclingo* at the time of this writing is still being developed. It currently does not allow the incremental introduction of variables to the program. This restricts supplying the program with non-ground rules through the incremental steps. Moreover, *oclingo* requires all the symbols used in the external input file be already introduced to the program in the initial grounding before the solver performs the first iteration of solving the program. One of the other features, that would have been useful in ranking the solutions presented by the system, is the optimization statements. This feature is available in the non-incremental version of the ASP-solver developed by University of Potsdam; however, it is still not implemented for the incremental versions.

5.1 Conclusion

In this work we highlighted use of ASP for high-level analysis in the rule-based dynamic domains. We investigated use of ASP as a component in a situation awareness (SA) system where we need to perform a large number of inference tasks in order to achieve a state of situation awareness. We have located our ASP component in a multi-layered situation awareness model called State Transition Data Fusion (STDF) model. The STDF model offers a modular model where the situational analysis tasks can be delegated into appropriate components. Pairing the STDF model with the *CoreASM* modelling framework provides a systematic approach to manage the complexity of the system through stepwise refinement of the system.

The *CoreASM* implementation manages the interactions between components performing situational analysis tasks. As shown in Figure 3.1, the lower-level computations in the first two layers are performed through *CoreASM* specifications in an imperative fashion. Higher level abstract analysis was accomplished using ASP in the impact assessment layer. We demonstrated how ASP offers a powerful and intuitive way of encoding the expert

knowledge in terms of rules. Having a system that is flexible and extensible is essential for domains that can have policy changes, which in turn will create rule changes in the system. ASP has a high flexibility to handle such changes.

The reactive answer set solver provides a seamless way of handling the history inside the ASP system. It enables simple means to discard the information which is no longer useful. The flexible means to manage the history from the previous time-steps makes it useful for dynamic domains.

5.2 Future Work

The response time of our system for the hand-input is reasonably short. However, we need to expose the system to the real world input in order to investigate how the accumulating history affects the response time of the system. One of the tracks for the future work is an extensive analysis of how the system compares to an alternative implementation where the history is handled using a database management system. This comparison can be made along quantitative and qualitative aspects. Quantitatively, the response time can be measured to examine how does the system scale as the input data grows. While in the qualitative aspect we can investigate the flexibility of the two systems with regard to policy changes, and the extensibility of the systems.

Additionally, further investigations to compare the approach using ASP with other logical languages such as *description logics* would be useful. This analysis would compare the languages based on expressiveness and the facilities provided by current implementations of their reasoner engines.

Another component that would be useful for future work is a component to automatically extend the rules given the domain expert knowledge. For arbitrary rules, this may be too difficult; however, it may be feasible for a restricted grammar of *if-then* rules.

Appendix A

ASP Program for Scheduling Marine Resources

A.1 Marine Resource Scheduler

Program A.1: Marine Resource Scheduler

```
#const max_duration=10.
#const max_vessel_id=10.
#const num_resources=3.

#base.

vessel_id(1..max_vessel_id).
resource(1..num_resources).
duration(1..max_duration).
preference(1..5).
time(0..10).

% preferred(priority).
% preferred(duration).

#cumulative t.

#external ext_request(I,R,D,t,P):
                        vessel_id(I) : resource(R) :
                        duration(D) : preference(P).
#external ext_now/1.
now(t) :- ext_now(t).

request_in_Q(I,R,D,t,t,P) :- ext_request(I,R,D,t,P).

request_in_progress(I1,R,D1,T1,t,P1) :-
                        request_in_Q(I1,R,D1,T1,t,P1),
                        not request_in_progress(I2,R,D2,T2,t,P2) :
                        vessel_id(I2) : duration(D2) :
                        preference(P2) : time(T2) : I2!=I1,
                        not request_in_Q(I3,R,D3,T3,t,P3):
                        vessel_id(I3) : duration(D3) : time(T3):
                        preference(P3): P1<P3, not preferred(duration).

request_in_progress(I1,R,D1,T1,t,P1) :-
                        request_in_Q(I1,R,D1,T1,t,P1),
                        not request_in_progress(I2,R,D2,T2,t,P2) :
                        vessel_id(I2) : duration(D2) :
                        preference(P2) : time(T2) : I2!=I1,
                        not request_in_Q(I3,R,D3,T3,t,P3): vessel_id(I3) :
                        duration(D3) : time(T3): preference(P3): D1>D3,
                        not preferred(priority).

request_in_progress(I,R,D-1,T,t,P) :-
                        request_in_progress(I,R,D,T,t-1,P), D>1.

request_in_Q(I,R,D,T,t,P) :-
                        request_in_Q(I,R,D,T,t-1,P),
                        not request_in_progress(I,R,D,T,t-1,P).

scheduled(I,t) :- request_in_progress(I,_,_,_,_,t,_).

#hide.
#show now/1.
#show scheduled/2.
```

Bibliography

[1] F. Baader, A. Bauer, P. Baumgartner, A. Cregan, A. Gabaldon, K. Ji, K. Lee, D. Rajaratnam, and R. Schwitter. A novel architecture for situation awareness systems. *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 77–92, 2009.

[2] M. Balduccini and S. Girotto. Formalization of psychological knowledge in answer set programming and its application. *Journal of Theory and Practice of Logic Programming (TPLP)*, 10(4-6):725–740, 2010.

[3] E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer-based systems. *Formal Aspects of Computing*, 19(2):225–241, 2007.

[4] G. Brewka, I. Niemela, and M. Truszczynski. Preferences and nonmonotonic reasoning. *AI Magazine*, 29(4):69, 2009.

[5] M.R. Endsley. Toward a theory of situation awareness in dynamic systems. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 37(1):32–64, 1995.

[6] R. Farahbod. *CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems*. PhD thesis, SIMON FRASER UNIVERSITY, 2009.

[7] R. Farahbod, V. Avram, U. Glasser, and A. Guitouni. Engineering situation analysis decision support systems. In *Intelligence and Security Informatics Conference (EISIC), 2011 European*, pages 10–18. IEEE, 2011.

[8] R. Farahbod, V. Gervasi, and U. Glässer. Executable formal specifications of complex distributed systems with coreasm. *Science of Computer Programming*, 2012.

[9] Fisheries and Oceans Canada; Canadian Coast Guard. Pacific - mcts: Vessel traffic services - canadian coast guard. http://www.ccg-gcc.gc.ca/e0003910, 2012.

[10] Cocco Gabriele. Train control system: An example of a system specification and simulation using asm and coreasm.

[11] M. Gebser, T. Grote, R. Kaminski, P. Obermeier, O. Sabuncu, and T. Schaub. Stream reasoning with answer set programming. 2011.

[12] M. Gebser, T. Grote, R. Kaminski, and T. Schaub. Reactive answer set programming. *Logic Programming and Nonmonotonic Reasoning*, pages 54–66, 2011.

[13] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental asp solver. *Logic Programming*, pages 190–205, 2008.

[14] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A users guide to gringo, clasp, clingo, and iclingo. *University of Potsdam, Tech. Rep*, 2008.

[15] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic programming*, volume 161, 1988.

[16] D.A. Lambert. Grand challenges of information fusion. In *Proceedings of the Sixth International Conference on Information Fusion*, volume 1, pages 213–220, 2003.

[17] D.A. Lambert. Stdf model based maritime situation assessments. In *Information Fusion, 2007 10th International Conference on*, pages 1–8. IEEE, 2007.

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

[19] V. Lifschitz. What is answer set programming? In *Proc. of AAAI*, pages 1594–1597, 2008.

[20] J. McCarthy, P. Hayes, and STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE. Some philosophical problems from the standpoint of artificial intelligence. 1968.

[21] Dwayne L. (US Coast Guard Tampa Bay Florida USA) Meeking. Improving waterway safety with cooperative vessel traffic service. http://www.kongsberg.com/en/kds/kncit/support/media/D4CC09E3A14B4A59A7D3BEEBD5BA736D.ashx, 2007.

[22] A. Mileo, D. Merico, and R. Bisiani. Support for context-aware monitoring in home healthcare. *Journal of Ambient Intelligence and Smart Environments*, 2(1):49–66, 2010.

[23] M. Nogueira and N. Greis. Application of answer set programming for public health data integration and analysis. *Availability, Reliability and Security for Business, Enterprise and Health Information Systems*, pages 118–134, 2011.

[24] J. Roy. Automated reasoning for maritime anomaly detection. In *Proceedings of the NATO Workshop on Data Fusion and Anomaly Detection for Maritime Situational Awareness (MSA 2009), NATO Undersea Research Centre (NURC), La Spezia, Italy*, pages 15–17, 2009.

[25] J. Roy. Rule-based expert system for maritime anomaly detection. In *Proceedings of SPIE*, volume 7666, page 76662N, 2010.

[26] J. Roy and M. Davenport. Categorization of maritime anomalies for notification and alerting purpose. In *NATO Workshop on Data Fusion and Anomaly Detection for Maritime Situational Awareness, La Spezia, Italy*, pages 15–17, 2009.

[27] P. Simons. Extending the stable model semantics with more expressive rules. *Logic Programming and Nonmonotonic Reasoning*, pages 305–316, 1999.