AN ABSTRACT MATHEMATICAL FRAMEWORK FOR SEMANTIC MODELING AND SIMULATION OF URBAN CRIME PATTERNS

by

Komal Singh

B.I.T, University of Delhi, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Fall 2005

APPROVAL

Name: Komal Singh

Degree: Master of Science

Title of thesis: An Abstract Mathematical Framework for Semantic Modeling and Simulation of Urban Crime Patterns

Examining Committee: Dr. Richard Vaughan
Chair

Dr. Uwe Glässer, Senior Supervisor

Dr. Martin Ester, Supervisor

Dr. Evgenia Ternovska, SFU Examiner

Date Approved: SEPTEMBER 8, 2005

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENCE

Abstract

Crime is not random. Criminologists contend there is predictable rationality and definite patterning behind urban crime. Conventional research for crime analysis is statistical and empirical in nature. However, with increasing complexity of the involved sociological system, empirical deduction is not sufficient; mathematical and computational models are needed for reasoning about system dynamics.

In this thesis, we posit a novel approach of computational modeling of urban crime patterns. By combining the Abstract State Machine (ASM) formalism with the Multi Agent System (MAS) modeling paradigm, we obtain an abstract formal framework for semantic modeling and integration of established theories of crime analysis. Such a firm mathematical foundation also provides a quintessential platform for constructing discrete event simulation models.

The framework can be applied for *predictive* and *explanatory* modeling of crime patterns. The virtue of this work is in its pioneering nature. It introduces an unprecedented, interdisciplinary research field of *Computational Criminology.*

to

my father, my god
my mother, my guardian angel

my brother, my philosopher
my fiancé, my sunshine

"Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius — and a lot of courage — to move in the opposite direction. "

— ALBERT EINSTEIN

Acknowledgments

This work could not have seen its consummation without Dr. Uwe Glaesser, my senior supervisor. I offer my enduring gratitude to him for his unflagging guidance, vision and support.

Special thanks to Dr. Patricia Brantingham for introducing me to the fascinating field of Criminology and making the idea of Computational Criminology a reality.

Acknowledgments are due to Drs. Eugenia Ternovska and Martin Ester for their valuable inputs.

Many thanks to Mona Vajihollahi for her insight and for helping me shape my bohemian ideas. Heartfelt thanks to Steven Bergner for his selfless contribution and fancy visualization for our mundane results.

Last, but not least, I am eternally grateful to my family and friends for their unwaivering love and moral support.

Contents

List of Tables

List of Figures

List of Specifications

Abbreviations and Acronyms

ASM	Abstract State Machine
ABM	Agent Based Modeling
ABSS	Agent-Based Social Simulation
AI	Artificial Intelligence
BDI	Belief Desire Intention
CBR	Case-Based Reasoner
DASM	Distributed Abstract State Machine
FAABS	Formal Approaches to Agent-Based Systems
FM	Formal Methods
MABS	Multi-Agent Based Systems
MAS	Multi-Agent Systems
MBR	Model-Based Reasoning

Application Specific

ADM	Agent Decision Module
SEM	Space Evolution Module
TSM	Target Selection Module

PART I

BACKGROUND

Chapter 1

Introduction

Crime is not random. Any crime consists of four dimensions -- the law, the offender, the target and the place [11]. Environmental Criminology is the study of the fourth dimension, the geography of the place, which is defined as the discrete location in time and space at which the other three dimensions intersect and a criminal event occurs. Criminologists have developed various theories of crime [10] that contend there is definite patterning in the *temporal* and *spatial* characteristics of physical urban crime.
The main theoretical theme argues that criminal events can be understood in the context of peoples' movements during the course of their everyday lives, and that criminals behave much like non-criminals most of the time. This implies there are a set of patterns/rules that govern the working of a typical real-life social system — composed of criminals, non-criminals, victims and targets, interacting with each other, in a given dynamic environment. The locomotion of people belonging to this system is influenced by the underlying urban landscape — city's land use patterns, street networks, transportation systems and typography. Furthermore, as chaotic as it may appear, there is predictable rationality that guides the victimization and decision-making process exhibited by criminals. Since there is definite patterning in crime, these patterns can be studied, stated and hence predicted.

In this thesis, we posit an abstract mathematical framework for semantic modeling and integration of established theories of crime analysis and prediction. This is obtained by combining the Abstract State Machine (ASM) formalism [6] with the Multi-Agent System (MAS) modeling paradigm [89]. The model serves as a platform for *predictive* (prescriptive)

and *explanatory* (descriptive) modeling of crime patterns.
The abstract framework serves as a ground model and provides a logical backbone for interfacing with diverse knowledge-based, and model-based systems through well-defined functions, and thus incorporating cross-disciplinary perspectives in an elegant fashion. The power of abstraction accompanied with step-wise refinement [7] provides a means to incrementally extend the model along different dimensions and units of functionality. Although *abstract*, the model is *complete* and *precise*, with respect to the given level of detail [5].
At the same time, as a secondary outcome, we obtain discrete event simulation models that serve as effective instruments for prediction and prevention of urban crime. These tools allow for experimentation and sample runs to reason about 'what if' and 'most likely' scenarios. They provide us with means of performing simulation-enhanced thought experiments aimed at improving our intuition and understanding about the modeled phenomenon.

Specifically, we focus here on *physical crime* in *urban areas* and model *spatial* and *temporal* aspects of criminal events, potentially involving multiple offenders and multiple targets. The scope of the model includes a broad range of crimes, ranging from mundane crime like robbery, car theft, burglary etc., to crimes of passion such as serial murder, homicide, rape etc. [8] [93] [83] [10].

This work thus introduces a novel, cross-disciplinary research initiative, broadly classified as *Computational Criminology* — spanning the fields of Modeling & Simulation, Formal Methods, AI, Algorithms, Criminology and Psychology. The core of this multi-disciplinary confluence is the ASM framework, that concerts with other fields to form the basis for a hybrid system (Figure 1.1).

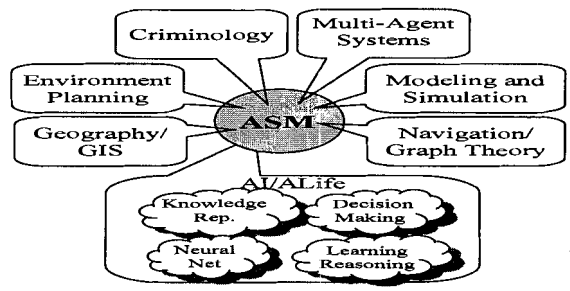

Figure 1.1: *ASM as the Core of a Multi-Disciplinary Confluence.*

1.1 Motivation

The primary impetus behind this work is the fact that personal security is of paramount importance. In the current day, post Cold-War era, significant interest is vested in areas of terrorism research, intelligence analysis, and security-related public policy. There is a pressing need for the study and development of advanced information technologies, computational sciences for national/international and homeland security related applications, through an integrated technological, organizational, and policy based approach.
To this end, our work in Computational Criminology pioneers in exploring the development of formal and computational models for crime analysis and prediction, and advancing the state of knowledge.

Conventional research for crime analysis is typically statistical and empirical in nature using methods that entirely rely on direct extrapolations from past data. Novel research directions [16], [56] however suggest a fundamentally different approach. Due to the increasing complexity, dynamics and intricate nature of the underlying sociological systems, empirical deduction is not sufficient any more; mathematical and computational models are needed for reasoning about most likely scenarios [17]. Although theories of crime are well established in their respective fields, the lack of a coherent and a consistent semantic framework for integrating the four dimensions of crime inhibits their applicability to real-life scenarios.
" Most research in criminology and criminal justice is structured in an attempt to replicate laboratory science by addressing simple issues...while assuming other things have been controlled. We have concluded that there is a strong need for criminological research that addresses complexity instead of attempting to control for it. [17]"
Thus, there is a need for a firm semantic foundation which is a prerequisite for the systematic construction of well defined computational models in order to overcome the limitations of purely statistical methods [56].

Finally, although the original definition of ASMs is intended to capture any kind of discrete dynamic system [57], their application to social systems is novel and unprecedented. This work thus gives us an interesting exploratory opportunity to apply ASMs beyond the traditional real of hardware and software systems.

1.2 Significance and Objective

The virtue of this work is in its pioneering nature [19] [20] [18] [21]. It is the forerunner in rendering the theoretical field of Computational Criminology [17], [16] a pragmatic and a tangible base, sound both from a computational and a criminological perspective. To the author's best knowledge, there has been no former published research in Computational Criminology, of the magnitude presented in this work. [56] posits a framework for crime analysis, however it is only theoretical and very preliminary in nature with no concrete results.

Mathematical modeling of crime serves multiple purposes. It has a direct value in law enforcement, in intelligence led policing, and in proactive crime reduction and prevention. For intelligence led policing, this model would make it possible to predict likely activity space for serial offenders for precautions and for apprehension. For proactive policing, modeling of crime makes it feasible to build scenarios in crime analysis and prevention, and provides a basis for experimental research allowing experiments that can often not easily be done in the real world [18].
Environmental criminologists could utilise this technique to test and refine theory, to anticipate consequences accruing from different intervention choices, and to provide informed policy guidance to crime control agencies. An agent-based simulation modeling technique will allow the study of the macro-level crime patterns emerging from the micro-level actions of many individual agents, behaving in accordance with rules derived from environmental criminology, human ecology, routine activities theory and situational crime prevention. [16].

The goal is twofold. First, to formalize the existing expertise and knowledge of the criminologists to derive an ASM ground model, that is *precise*, *flexible*, *understandable*, *complete* and *operational* [5]. Such a model is used for semantic modeling that greatly simplifies the systematic integration and validation of crime patterns and theories, and helps gain a better understanding of complex social system aspects. Second, such a solid mathematical foundation provides a quintessential platform for subsequent refinement into and construction of discrete event simulation models. Simulation models can be used for experimental validation and verification. They are also effective instruments for prediction and prevention of crime.
Although the model currently focuses on *spatial* and *temporal* aspects of *physical* crime

in *urban* environments, it is abstract and general, and in principle scalable for different types of crime and also different levels of spatial analysis. In addition to conventional crime being already captured, the model can be potentially extended to simulate patterns of non-conventional crime like corporate crime, cyber crime, intrusion detection. In principle, it can also be applied to simulate patterns at different levels of spatial aggregation viz: airports, malls, downtown; within cities, provinces; and between countries, geopolitical crime.

The target audience of the system thus developed can be *criminologists* using it for deeper understanding of crime patterns, verification of hypothetical theories; *modern policing agencies* using it for geographic profiling of criminals, inversion of crime to identify suspects; *city planners* using it for for effective urban planning by taking geography of crime into account.

1.3 Thesis Organization

For sake of perspicuity, the ASM model and hence the thesis is divided into five parts. Part I and Part V present the background and conclusions; Parts II, III, and IV discuss the ASM model at different levels of abstraction in an increasing order of detail and complexity viz: *Abstract Model, Refined Model and Executable Model* respectively.

Part I of the document is intended to provide the reader an overview of the problem domain and the formal modeling technique used. Chapter 1, the current chapter, introduces the research work being presented and discusses the motivation and significance of the work. Chapter 2 gives a brief introduction to the fascinating field of Environmental Criminology and explains the problem domain. Chapter 3 provides an introductory knowledge of Basic and Distributed Abstract State Machines (ASM) and investigates the technique for high-level system analysis and design.

Part II of the document describes the ASM ground model at the first level of abstraction. We call the model at this preliminary level of detail the *Abstract Model.* There are intermediate hierarchical levels of refinement of the Abstract Model within this part, however, we still group these refinements under the Abstract Model. Chapter 4 provides an overview of the model. Chapter 5 talks about techniques for modeling social systems, discusses the predominant view of Multi-Agent Systems, and provides an overview of the adopted modeling paradigm. Chapter 6 details the approach we take on representation of the subjective

and objective environment. Chapter 7 provides an explanation and corresponding ASM specification of the architecture of the model and its constituent modules.

Part III of the document describes the ASM ground model at the intermediate layer of abstraction. We call the model at this level of abstraction the *Refined Model.* The refined Model is obtained by applying a refinement step to the Abstract Model of Part II. Chapter 8 gives an introduction to the mechanism of Case-Based Reasoning, and further describes the proposed learning and reasoning mechanism. Chapter 9 provides an overview of the shortest path problem and further describes the proposed shortest path algorithm.

Part IV of the document describes the model at the final layer of abstraction. We call the model at this level the *Executable Model.* The Executable Model is obtained by applying a refinement step to the Refined Model of Part III, and provides formal executable semantics of the ASM ground model. Chapter 10 starts with introducing the Abstract State Machine Language (AsmL) that is used to derive the Executable Model, provides an overview of our AsmL model, and discusses the visualization we develop. Chapter 11 is devoted to experimental validation of the AsmL model.

Part V of the document provides a closure. Chapter 12 gives the conclusions of the work, talks about challenges met and main contributions; finally Chapter 13 discusses challenging opportunities for future research and expansion of the current work.

Chapter 2

Environmental Criminology

2.1 Introduction

Crime is a complex multi-dimensional event; albeit not *random*. There are four dimensions of crime — a legal dimension, an offender dimension, a target dimension and a spatio-temporal dimension [11]. Each dimension has many research vectors, and these are often similar. To understand crime, those dimensions must be understood and interpreted against a complex historical and situational backcloth of social, economic, political, biological and physical characteristics [12].
Environmental Criminology is a discipline devoted to the study of the fourth dimension – the spatio-temporal aspects of crime — that can be seen as a discrete location in time and space at which the other three dimensions intersect and a criminal event occurs. It attempts to analyze the role of time and space in the shaping and distribution of criminal events. First issued in the works of urban planners, where the authors propounded that the alteration of urban design and urban architecture would exacerbate or abate crime, it has become a field of study in its own right.

Environmental Criminologists set to answer questions such as [17]:

- How do crime patterns change with the introduction of new motorways connecting two cities, or transit system failing to operate on a given day.
- How is crime attracted to areas around large shopping centres?

- How do urban setting create crime attractors or crime generators?
- Why does increased enforcement displace crime in one city but abate it in another?
- Why is crime high along one main road and low along another?
- How would crime patters change with increased mobility or migration?
- Why does regulation of pub hours have different impacts in different cities?

The first stance taken in Environmental Criminology is that criminal events can be understood in the context of peoples' movements in the course of their everyday lives. Most crimes are observed in the areas criminals are highly active in and highly familiar with; most victims are victimized in and around areas they are most familiar with. This comfort zone of people is termed as their *awareness space.* While the above line of reasoning seems counter-intuitive, there is sufficient theoretical and empirical evidence in support of same. Section 2.2 investigates this line of reasoning in more detail.

The second major stance is that there is sufficient predictability and rationality behind an offender's choice of a target. The location of crime is determined through a premeditated decision process of the offender, supplemented by subjective perceptions of environment that separate good criminal opportunities from bad risks -- targets located along highly accessible street networks attract crime. This in general can be termed as *target templating*, and is further explored in Section 2.3.

The third most important stance is that *movements* of people are influenced by the *underlying urban landscape* in concert with their subjective *perceptions* of the environment. The land use patterns, the structure of street networks, the transportation systems, the typography, traffic and transit patterns all play their respective roles in shaping the movements of people. In addition, the way people measure and filter this absolute knowledge depends on a set of socio-cultural and behavioral beliefs. Thus people have varying perceptions of the same geographic reality. This behavioral aspect of criminology is probably the lesser developed one. We set to explore the factors that determine the movements of people in Chapter 9 and the role of perception in Chapter 6.

Thus, environmental criminologists view the learning and decision-making process for criminals to be much the same as that for non-criminals. Criminals are rarely criminal all of the time. Everyone has a degree of criminal potential; what differs is the level of criminal

propensity — more criminally predisposed agents will respond to observed crime opportunities more frequently than people with low crime potentials. What requires modeling is the process by which agents move from node to node, and when, if at all, do they act upon an observed criminal opportunity.

There are three levels of spatial analysis in current studies of environmental criminology viz *macro-analysis, meso-analysis and micro-analysis* [12], [10].
Macro-analysis is the highest level of spatial aggregation, and analyzes spatial distribution of crime *between* countries, provinces or cities. Countries are seen as passing through stages of development and interacting with each other, along which comes a host of demographic and economic change that results in crime. *Meso-analysis* is the intermediate level of spatial aggregation, and studies crime distribution *within* larger regions such as provinces and cities. *Micro-analysis* is the lowest level of spatial analysis and involves studying crime *within very specific areas*, such as small part of a city.
Spatial patterns in crime differ depending on the level of analysis selected, although the principles remain the same. The work presented in this thesis, can be best described as belonging to the third level of analysis, i.e the micro-level.

The principles of environmental criminology can be used to explain the very nature and occurrence of a broad range of crimes — ranging from mundane crime like robbery, car theft, burglary etc., to crimes of passion such as serial murder, homicide, rape etc. [8] [93] [83] [10]. While conventional research in criminology is best applied for the study of physical crime of spatial nature, one should be cognizant of the fact that the notion can potentially be extended to other non-conventional modes of crime, such as white-collar crimes, corporate crime, cyber crime, intrusion detection etc. This line of contention, however, needs more thorough analytical research.

2.2 Activity and Awareness Space

The task of simulating the movement of a hypothetical offender is complex, but the main elements can be shown here. Criminals, to the extent, that they live in everyday society are bound by normal limitations on human activity, shaped by dictates of work, family, sleep, food, entertainment, and so forth. They all live within the confines of their own knowledge

and experiences and habits [25]. Each person is tied to what is called 'Anchor Points' and there are three main classes of these anchor points or activity nodes : home, work and recreation [42]. During the course of everyday life, to carry out everyday routine, the agent then travels between these nodes using familiar pathways; the more often an area is visited, the more knowledge the agent will gain regarding the immediate surroundings for both the nodes and the pathways connecting them [15].

Based on this movement and a priori knowledge, a person develops a *mental map*, which is a representation of the spatial form of environment that an individual carries in his or her mind. Mental maps of familiar areas such as neighborhoods or cities, are formed from a distillation of the particular transactions a person has with his or her surroundings [25]. These cognitive images are the result of the reception, coding, storage, recall, decoding, and interpretation of information [10].

The mental map of a person is built around their *awareness space*, which is composed of all locations about which a person has knowledge above a minimum threshold level even without visiting some of them. This awareness is developed by knowledge-exchange with other people, past experiences and current events. The awareness space is derived from a number of factors — primarily based on the activity space described below — such as strength of activity space, mobile visibility, accessibility from activity space, willingness to leave activity space, attractiveness of adjacent areas, speed and model of travel etc.

Within this generalized awareness space, a specialized knowledge is formed by direct experience, knows as the *activity space*. The activity space can be defined as the space in which majority of an individual's activity is carried out, and contains those places and connecting routes which comprise a person's habitual geography on a daily basis [10]. "Where we go depends upon what we know...what we know depends on where we go" [25]. The activity space contains only that sub-set of the awareness space that is visited and traveled by a person on a regular and a current basis. Typically, people make natural and intuitive choices while trying to decide on a path to take, such as trying to minimize time and distance of travel, moving toward the destination instead of away from it, preferring major roads to minor roads, using familiar paths as compared to unknown paths etc. Based on these intuitive decisions, one can model the temporal and spatial movement patterns of people.

Both the activity and awareness space are dynamic in nature. Spaces grow with increased

movement and as new locations are discovered or new information gathered. At the same time, the spaces may shrink if the person has not visited or recalled these spaces in a given interval of time, i.e the person's memory fades over time unless refreshed. However, after an initial learning period in a new location, the activity and awareness spaces become relatively fixed. Furthermore, the spaces vary in their strength — highly active paths, such as those located on daily home-work schedules, have a higher intensity, than those paths taken rarely, such as a yearly hiking trip. Figure 2.1 explains pictorially the formation of activity and awareness space.

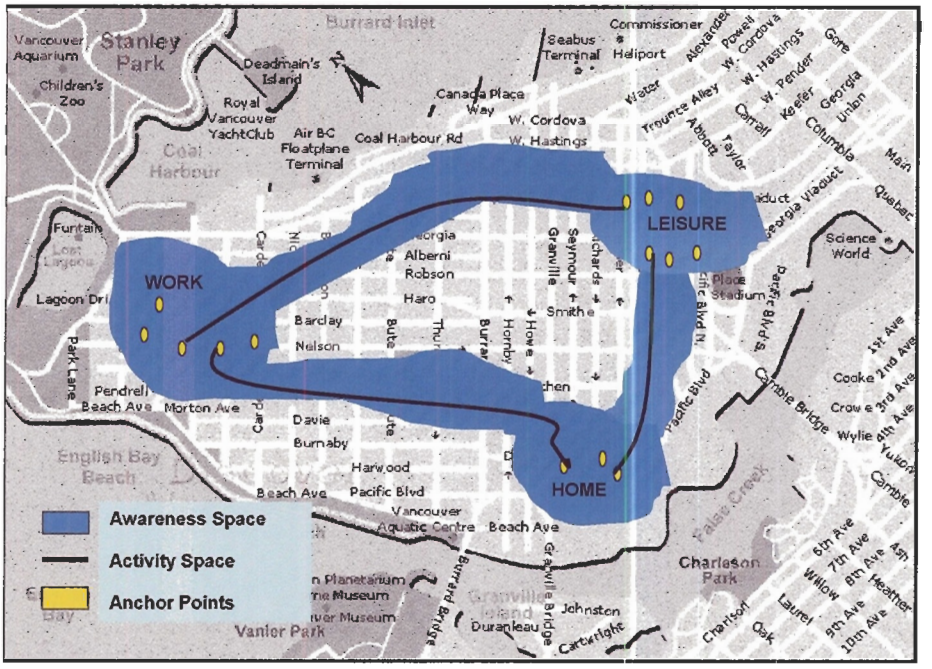

Figure 2.1: *Formation of Activity and Awareness Space.*

The daily routine of people vary with their demographic backgrounds, and as such the awareness spaces tend to be different for different people. Demographic factors such as age, race, gender, profession, economic level etc. play a role in carving out the activity and awareness space. For example: a person between the age of 60-70 will be a lot less mobile than a teenager, a traveling salesman will have a larger activity/awareness space than a housewife.

An example in point is needed. Assuming that our criminal agent moves about in space much like most people, the agent is tied to at least three main classes of activity nodes: home, work and recreation. The agent travels between these nodes using familiar pathways. If the agent starts his or her day at home, and then travels to either work (or school), for example, he or she will typically take the most direct and most easily navigated route. Along the way, the agent will take notice of a range of phenomena in his activity space. For example, he might see a favorite coffee house, or take notice of a particular shopping opportunity along the way to his work node. Even if the agent does not stop and interact specifically with potential activity sites, he will often remember such sites the next time he wants to purchase a cup or coffee or to patronize a particular business. The same learning process applies to the travel paths to and from other significant activity spaces.

The activity and awareness spaces of people are of crucial significance. These represent the 'comfort zones' of individuals in which they are most likely to direct all their actions. Thus, it is in these spaces that potential opportunities are observed by criminally-disposed people and subsequently acted upon [15].

2.3 Target Selection

As chaotic as the process of target selecting seems, there is predictable rationality behind it that guides the process; this is called *target templating* [14].
We use the term *opportunity space* to refer to the space of potential targets; this space is typically an objective reality and is universally defined for all criminals, different criminals however might have different perceptions of it. The term *crime occurrence space* refers to the space where targets are highly probable of being victimized or selected by a criminal; this space is typically agent-specific or could be an aggregate.

The *Rational Choice Hypothesis* developed by Cornish & Clarke [27] [30] suggests that the decision to choose a target is a rational one based on a rough cost-benefit analysis assessing the risk involved and the potential payoff. Offenders rationally assess all information about potential crime and make a rational choice based on an assessment of consequences, combined with the offender's background (intelligence, personality, upbringing) and situational factors. The important point is that this hypothesis views the committing of a crime as a

series of decisions and processes made by the offender in the commission of that crime.

The same theme of predictability runs in the *Routine Activity Hypothesis* [28] [42], which sees crime as convergence in space and time of three minimal elements : 1) motivated offenders, 2) suitable targets, and 3) absence of a capable/motivated guardian. This theory introduces the crime triangle (also called the problem analysis triangle (PAT)), centered around offender, target and place, that forms a tool in crime analysis. The focus is on the nature of everyday patterns of social interaction to analyse crime: the routine activity of leaving the home unattended, absent a guardian, during the workday increases probability of crime.

The *Model of Crime Selection* as proposed by the Brantinghams [9] [10] uses the above concepts of *opportunity, motivation, and decision-making* and ties it with *mobility and perception.* It contends that crimes are most probable to occur in those areas where the awareness space of the offender intersects with the perceived suitable targets, i.e *the crime occurrence space is the intersection of the awareness space and the opportunity space* (Figure 2.2). Within this crime occurrence space, the offender then uses target templating — assesses 'cues' from the environment and targets, weighs opportunities and risks, makes rational choices and finally chooses specific targets for victimization. Once a template is established, it becomes relatively fixed and self-reinforcing.

Brantinghams' Model of Crime Site Selection views target selection as an information processing model given as propositions — *Proposition I:* Given the motivation of an individual to commit an offense, the actual commission is the end result of a multi-staged decision process. In case of high emotional involvement, the process involves a minimal number of steps; in case of low emotional involvement, the process may include more deliberative steps. *Proposition II*: The environment emits many signals or cues about its physical, spatial, cultural, legal and psychological characteristics. *Proposition III*: The motivated criminal uses these cues, learned through experience or social interactions, to locate and identify targets. *Proposition IV*: As experiential knowledge grows, motivated criminals learn which individual cues are associated with 'good' victims. These cue sequences (spatial, temporal etc) can be considered a *template* for target selection; potential targets are compared against this target template and either rejected or accepted depending on consequences. *Proposition V*: Once the template is established, it become relatively fixed and self-enforcing. *Proposition VI*: Because of multiplicity of targets and victims, many potential crime selection templates

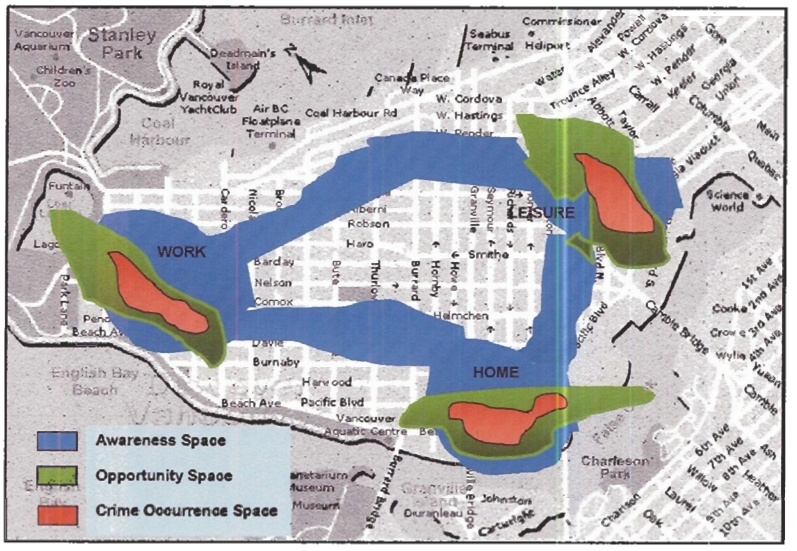

Figure 2.2: *Crime Occurrence Space as the Intersection of Awareness and Opportunity Space.*

could be constructed; however individual templates have similarities which can be identified.

Based on the above Model of Crime Site Selection, the Brantinghams further develop different 'cases' of spatial distribution of offenders and targets, ranging from single to multiple, from uniform to non-uniform, and show the search areas obtained in different cases [13].

Finally, demographic factors play a role in determining the 'criminal propensity' of offenders. Offenders with a high propensity are likely to commit crime more often. Factors such as high drug use, frequent nights out, high poverty, etc. exacerbate crime.

A brief example in point is needed to illustrate the rationality behind crime selection. A typical crime in large cities is that of burglary, otherwise known as break and enter, or "B&E". In the large majority of cases, for the agent to be aware of a potential target, the site must be located within the agent's activity space - which, in turn, is defined by the set of common activity nodes, as discussed earlier. The burglar would travel from Node 1 (home) to Node 2 (work), as his routine may require. Along the way he recognizes a residential building that is suggestive of a 'good' target, as it 'fits' within that particular agent's crime template. Research on burglars [80] [81] [40] suggests that variables such as

property value, lack of occupants or potential witnesses capable of intervening, and obvious entry opportunities, all form 'cues' [39] from which the agent assesses quickly to determine if either the criminal event will be attempted, or at least investigated further.

Chapter 3

Abstract State Machines

The abstract framework presented in this thesis is based on the Abstract State Machine (ASM) formalism. This chapter is intended to provide the reader a lucid working definition of the theory of Abstract State Machines.

Abstract State Machines (ASM) are a universal mathematical approach for modeling *discrete dynamic systems.* Based on first order logic, abstract state machines can be used to specify, analyze and construct complex systems at their natural levels of abstraction, in a simple yet elegant manner. With an operational semantics, and a light-weight agile methodology, it is universal in its appeal for modeling a plethora of systems.

Propounded by Yuri Gurevich in the early 1980s, whence they were called Evolving Algebras [57], they were an attempt to bridge the gap between formal models of computation and practical specification methods. The ASM technique has now evolved into a mature and a pragmatic technology that provides a rigorous approach for modeling systems with precision, consistency and unambiguity.
ASMs have been successfully deployed in a wide array of applications — verification of algorithms and protocols, specification of hardware and software architectures (DLX, PVM; COM), modeling of real-time distributed systems, semantic modeling of languages (SDL, VHDL, UML), verification of compilers (Java, Occam), software engineering etc [6].

We begin with Section 3.1 that explores the applicability of ASMs for systems-engineering and *high level system analysis and design.* Section 3.2 explains the theory of Basic ASMs,

and examines the constructs of *parallelism* and *non-determinism.* This is followed by an investigation of Distributed Abstract State Machines (DASM) in Section 3.3, that are widely used for modeling *distributed, real-time, concurrent* systems. Finally, Section 3.4 enumerates some commonly used notational conventions.
For an in-depth study, and a mathematical investigation of the subject, the reader is referred to popular literature [58] [57] [6].

3.1 High-Level System Engineering with ASMs

The history of Software Engineering stands witness to the fact that more often than not software systems run over-time and over-budget, are inconsistent and incomplete, and almost never capture user needs fully. The formidable task of Requirement Engineering [76] is quintessential to good software design and consequently quality software construction. However, unlike classical engineering disciplines, the field of software engineering is relatively open-ended, and hence there exists no silver bullet [24] that can attack the very essence of software development and ensure the right software. Fortunately, the realm of Formal Methods posits a solution to one very important aspect of successful software development — requirements engineering in a consistent, complete and unambiguous manner [4].

ASMs are applied in the same spirit, to bridge the gap between informal user requirements and formal software construction; in other words turning English into Mathematics, while preserving lucidity and perspicuity of the model. ASMs provide an accurate semantics for a formalism without providing the hassle of understanding the mathematics behind the formalism. The basic definition of ASMs accompanied with the three building blocks of *Abstraction*, *Ground Model Construction* [5] and *Refinement Techniques* [7], make it a universal and a pragmatic tool for systems engineering.

In this Section we seek to explain the suitability and applicability of ASMs for high-level system analysis and design: Section 3.1.1 and 3.1.2 examine the concepts of ASM ground model construction and refinement techniques respectively; Section 3.1.3 explores the role of ASM as a software development methodology.

3.1.1 Ground Models

As mentioned above, the task of Requirements Engineering [76] is quintessential to good software design. The *process of formalizing* the requirements is what can be termed as ground model construction. It is clearly the problem of turning English into Mathematics, while preserving the ease of understanding of natural language. Ground models *freeze* or *ground* the requirements, so that one may have a blueprint of the desired system.

Boerger [5] identifies three broad problems that all ground models should address. Firstly, ground models should provide a suitable formalism to mediate between the problem domain and the world of models; this is the *language of communication* and should be easy to understand by people of both the domains. Secondly, the problem of *verification* should be addressed; verification is the process of proving that you are building the 'system right'. Thirdly, the *validation* problem, whether you are building the 'right system' should be addressed.
Further, the author enumerates five *intrinsic properties* that all ground models should have. Firstly, they should be *precise yet flexible.* Secondly, they should be *simple yet concise.* Thirdly, they should be *abstract yet complete.* Fourthly, they should be *validatable.* Finally, they should have a *precise semantic foundation.*

ASM Ground Models satisfy all of the above desired characteristics. Based on a mathematical foundation, they provide a precise semantics. With a lucid syntax, they provide a simple and easy to understand formalism. This can be used to build specifications that are concise and flexible. Using different functions, one can build strong abstractions, and yet a complete model. It solves the verification problem by allowing one to use symbolic model checking, type checking etc. It also solves the validation problem by its operational character that can be used to build executable models or draw mental simulations.

3.1.2 Refinement Techniques

Refinement goes hand-in-hand with the inverse process of abstraction and plays a central role in incremental system development. With respect to other refinement techniques, the ASM refinement method is rather informal. It is not based on any concrete principle, and thus can be customized according to the need of the application. It should however, meet

the following principle of substitutivity:
Principe of Substitutivity: it is acceptable to replace one program by another, provided it is impossible for a user to observe that the substitution has taken place [32].

There can be two types of Refinements, viz: *Horizontal Extensions* and *Vertical Extensions.* Horizontal extensions provide new modules with new functionality to the existing application. Vertical extensions build upon existing functionality by elaborating the corresponding data structures and/or rules.

In addition, the ASM refinement method provides three types of refinement patterns. These are discussed below, as adopted from [7].

Conservative Refinement
Conservative refinement or conservative extension as it is widely called, is a type of incremental refinement. In support of modular system design, this can be used to introduce brand new functionality to the system. An example would be extending an abstract Java Virtual Machine with the functionality of exception handling.
For conservative refinement, a *new machine* is defined with corresponding behavior. The new machine is executed with the triggering of a *condition for new case.* At the same time, the *old machine* can be executed by negating the condition of the new case.

Procedural Refinement
Procedural refinement, or submachine refinement as it is widely called, is performed by replacing a given rule (or a submachine) by another rule (or a machine). One may use multiple submachines to achieve this affect, where the behavior of applying multiple submachines is abstracted away and seen as only one step.

Data Refinement
Data Refinements are those where abstract states and rules are mapped to concrete ones such that the result of applying concrete rules to concrete types is the same as applying abstract rules to abstract types. One such example is *Instantiation*, where the ASM rules remain unchanged and abstract functions are refined further.

3.1.3 Agile Development

Abstract State Machine provide an avenue for *agile software development.* Agile software development emphasises the need for being *adaptive* rather than *predictive*, being *people-oriented* rather than *process oriented*, *less document-oriented* and *more code-oriented* [47].

With the concepts of abstraction and refinement techniques, ASMs provide a means for *iterative development* as opposed to traditional waterfall-like methodologies. This style of development is most suitable for addressing the problem of requirements creep and being adaptive in nature.

Beginning with an abstract ground model (Section 3.1.1), and using proper step-wise refinement (Section 3.1.2) one can relegate ground models to executable code. This by itself provides a disciplined process of software development accompanied with verification and validation. As a by-product, the formal specifications can be used as valuable system documentation. Figure 3.1, adapted from [6], summarizes how ASM models are used in the hierarchical design and construction of software systems.

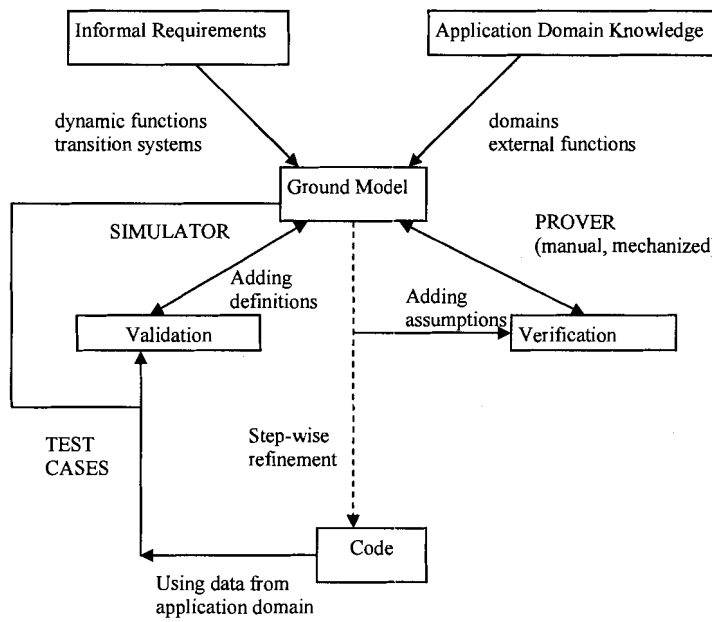

Figure 3.1: *The Hierarchical Software Development Process.*

3.2 Basic Abstract State Machines (ASM)

Basic ASMs can be seen as the definition of ASMs in their most original form. The intuitive understanding is to view them as *pseudo-code* over *abstract data structures.* A basic ASM is a single agent machine, as opposed to its DASM counterpart which can be multi-agent and asynchronous. Basic ASMs are endowed with potentially unlimited non-determinism and parallelism. The sequential ASM thesis contends that any sequential algorithm can be simulated at its natural level of abstraction by an appropriate ASM [59].

A basic abstract state machine has a *signature* V, which is nothing but a collection of *function* names. The machine transitions over abstract data structures called *state* S of the ASM, which in mathematical terms is nothing but a first-order algebraic structure. The machine is associated with a *program* P, which is a set of so called *transition rules.* Finally, the machine comes equipped with an initial state S_i.
Thus, a basic ASM can be seen as a tuple $[V, P, S, S_i]$.

Every ASM signature[1] is assumed to contain the static functions *undef, true, false.* The functions of signature V are called *basic functions.* Each function has an *arity* which is the number of parameters the function takes. *Relations* are similar to functions, except that they always map to either true or false. The default value for basic functions is undef and false for basic relations. Constants are represented as nullary functions.

A state S for the signature V consists of a non-empty set X, which is the *superuniverse* of V, together with the *interpretations* of the function and relation names of V. If f is an n-ary function name of V, then its interpretation in state S is denoted by f^S, which is a function from X^n to X. The superuniverse X of the state S is denoted by $|S|$ and is also called the *base set* of a state. Function names in a given state are interpreted as *total functions*, however, every total function can be viewed as *partial* w.r.t *undef*.

A basic *Program* P is a basic rule without free variables.
A *pure run* of P is a sequence $(Sn : n < i)$ of states of vocabulary V, such that each S_{n+1} is obtained from S_n by firing P at S_n. A *pure* run also called *internal run* is one that is not affected by the environment, as opposed to an *interactive run.* The notion of a run is similar to that of a *state transition* in classical systems. An ASM step in a given state

[1]The terms Signature and Vocabulary are used interchangeably.

executes all updates sets of all transition rules *simultaneously.* A legal *move* yields the next state, whereas an illegal move halts the machine.

3.2.1 Transition Rules

The basic language of transition rules is very modest and even minimal in some justifiable sense. There are four basic rules viz. *update rule, conditional rule, block rule, and import rule.* The only primitive transition rule is the *local function update*; other rules are variants and extensions of this rule [57].

The **Update Rule** has the form:
$f(t_1, ..., t_r) := t_0$
where, f is the name of a basic dynamic function, r is the arity of f and every t_i is a closed term. The pair $(f, (t_1, ..., t_r))$ is called a *location*, and the value $f(t_1, ..., t_r)$ is called the *content* of the location in given state S.
An *update* is a pair (l, v), where l is a location and v is a term. An *update set* is a set of such updates. Firing an update(set) has the semantics of replacing the content of location l with new value v. With the in-built parallelism, an update set may have rules, that update the same location at the same time with different values. This produces a *clash* and the update set is said to be *inconsistent.* For a consistent update set, firing the set produces a new state with the same superuniverse, but different interpretations of dynamic functions.

The **Conditional Rule** has the form:
if b **then** R1 **else** R2 **endif**
where, b (the guard) is a boolean valued predicate, and $R1$, $R2$ are updates.
The semantics of an update rule are equivalent to a *guarded* update. If b evaluates to true on the given static algebra, then perform $R1$; otherwise perform $R2$. Simplified version of this form, need not use the else clause or endif explicitly.

The **Block Rule** has the form:
do-in-parallel
R1
R2
where, $R1$ and $R2$ are update rules.
The semantics of such a block rule is equivalent to executing the rules $R1$, $R2$ simultaneously,

such that the order is immaterial. However, if the block rule produces an inconsistent update set, the machine halts. Simplified versions of the block rule, need not use the do-in-parallel explicitly.

The **Import Rule** has the form:
import x **do**
P
where, x is an element of the *reserve* and P is some statement.
The semantics of import rule is to select non-deterministically an element x from the *reserve*, delete it from the reserve, and execute P. This allows for dynamic allocation of resources during program runs. There are a few syntactic variations of the import rule.

3.2.2 Parallelism

At a high-level of abstraction, it is highly desirable to abstract away from sequentiality where it is irrelevant. The construct of parallelism in ASM extends the two concepts that time is sequential, and that only a bounded amount of work is performed in a step.
One may introduce agents that perform a substantial amount of work in a single step. A step may involve numerous parallelism; such work, in principle may be executed by several auxiliary agents executing in parallel. Nevertheless on a natural level of abstraction of an algorithm, such work is accomplished by a single agent, and those auxiliary agents are invisible [58].

The parallelism is expressed with the following syntax:
for all $x \in X$ **with** $f(x)$
R(x)
where, x is a variable, X is a domain, f is a function defined on x and R is a rule.
The semantics of the $forall$ construct is that for every x that satisfies the condition given by f, execute the rule R. There can be multiple rules, that are executed simultaneously, where the order of execution is immaterial. Thus, one may perform potentially unbounded work in a single step.

3.2.3 Non-Determinism

The phenomenon of non-determinism allows one to model scenarios that are not necessarily under an algorithmic control or are irrelevant at a high level of abstraction. Non-determinism may be modeled by using *external functions* (explained later), or by using the in-built ASM construct of *choose*.

The syntax of the choose construct is as follows:
choose $x \in X$ **with** $f(x)$
R(x)
where, x is a variable, X is a domain, f is a function defined on x, and R is a rule.
The semantics of such a construct is to execute the rule R for *any arbitrary* x that satisfies the property f.

3.2.4 Classification of Functions

In support of principles of modularization, information hiding, data abstraction and separation of concerns, the ASM method exploits the following distinction among the types of functions and locations [6].
An ASM M may have *static* functions — those that never change during a run of M, or *dynamic* functions — those that may change during a program run. Static functions are defined by the initial state of the ASM. Static and dynamic functions can be thought of as constants and variables of programming languages respectively.

The dynamic functions are further divided into four sub-categories[2].
Monitored or *In* functions are those that are updated by the environment only and not by the rules of M, but can be read by M. The monitored functions provide a strong abstraction mechanism and can be used to model irrelevant details or details not under the control of the model. These functions can be thought of as *oracles*, that given a set of arguments, magically provide the desired result. The oracle need not be consistent and may give different results for the same argument at different times. The seeming inconsistency may be quite natural. However, the oracle should be consistent during the execution of any one step of the program [57].

[2]An additional term *External* is used to refer to functions that are either static or monitored.

Controlled functions are dynamic functions that are updatable by and only by the rules of M. These are also known as *Internal* functions.
To describe a combination of internal and external control of functions, we have *Interaction* or *Shared* functions. Such functions are updatable by the rules of M as well as by the environment, and can be read by both. The concept of monitored and shared functions allows one to separate the computation concerns from communication concerns [6].
Finally, *Out* functions are dynamic functions which are updated, but not read by M. Conversely, they are read, but not updated by the environment. These functions allow the machine to pass information to the environment or to other agents in the environment.

Another taxonomy of functions is that of *derived* and *basic* functions. Basic functions come with the signature of an ASM by default. Derived functions are dynamic auxiliary functions which have a computational definition, and can be seen as storing some precomputed results. They are not updated by M or the environment, but can be read by both.

3.3 Distributed Abstract State Machines (DASM)

Distributed Abstract State Machines (DASM) extend Basic ASMs to incorporate the notion of multi-agent computations — a scenario, where multiple agents, each equipped with their own set of states and rules, run in parallel, independent of each other, but possibly interacting with each other.
The agents may be *synchronous* in their behavior or *asynchronous*. In case of synchronous systems, an implicit global system clock is used to orchestrate computations. Synchronous ASMs support modularity for the design of large systems. Asynchronous ASMs support the design and analysis of distributed systems.

Semantically, a distributed ASM D is composed of:

- A finite set of agents, called $AGENTS$. $AGENTS$ is a dynamic function.
- Each agent a is associated with a program P_a. The function name $MODULES$ or $PROGRAMS$ is used to represent the set of all such programs, which is a static nullary function name.

- Vocabulary V, which is the union of the signatures of each constituent component.
- Initial State S_0 which is a collection of initial states.

The DASM D has a *global state* and the agents interact with each other by sharing locations of this global state. Each agent a has a *view* V_a of the global state. Further, a static nullary function *Self* is used by agents for self-identification.
An agent a can make a *move* at state S by firing $Program_a$ at $View_a(S)$; to perform a move of agent a, fire
$Updates(a, S) = Updates(Program_a, View_a(S))$

Runs of a distributed ASM are defined in the following section.

3.3.1 Concurrency and Coherence

In case of Distributed ASMs, a problem may arise of the possible incompatibility of moves due to different data, time clocks, duration of executions [6].
Hence, the *run* of a DASM D, as quoted from [58], is defined as a triplet (M, A, σ) satisfying the following four properties:

1. "M is a partially ordered set, where all sets $\{y : y \leq x\}$ are finite.
   Elements of M represent *moves* made by various agents during the run.
2. A is a function on M such that every nonempty set $\{x : A(x) = a\}$ is linearly ordered.
   $A(x)$ is the agent performing move x. The moves of any single agent are supposed to be linearly ordered.
3. σ assigns a state of D to the empty set and each finite initial segment of M; $\sigma(\theta)$ is an initial state.
   $\sigma(X)$ is the result of performing all moves in X.
4. **The Coherence Condition**: If x is a maximal element in a finite initial segment X of M and $Y = X - \{x\}$, then $A(x)$ is an agent in $\sigma(Y)$ and $\sigma(X)$ is obtained from $\sigma(Y)$ by firing $A(x)$ at $\sigma(Y)$."

This definition produces two corollaries: (1) All linearizations of partially ordered moves of X(a finite initial segment) yield runs with the same final state. (2) A property holds

in every reachable state of a run R if and only if it holds in every reachable state of every linearization of R.
The above scheme is as liberal as possible, and thus can be instantiated by any well-defined synchronization mechanism.

3.3.2 Reactivity and Real-Time Behavior

ASMs are typically *reactive* systems as opposed to *transformational* systems.
The provision of different categories of functions (see Section 3.2.4) allows one to model different types of reactive behavior at desired levels of abstraction. These well-defined functions act as interfaces for communication to-and-fro a machine and its operational environment (or other machines). Monitored functions can be used to send information from environment to the machine. External functions can be used to delineate a system from its operational environment. Internal functions can be used to communicate information from the machine to the environment.

ASMs operate with a *discrete time* notion. Continuous time is mapped (by using sampling or other techniques) to discrete intervals.
Real time behavior imposes additional constraints on DASM runs requiring that the agents react instantaneously. An assumption is made that all actions take place in *atomic* time. Timing aspects are simulated with an *abstract* global system clock. A monitored function *now* acts as an oracle that provides the current time. The now function returns values from an abstract universe TIME.

3.4 Notational Conventions

The following notational conventions are used for ASM specifications, for improved readability:

- Agent Names start with a block letter. The individual words also start with block letters and are separated by an underscore _. The rest of the letters are also written in upper case (e.g. SEM, AGENT_DECISION_MODULE).

- Program names are derived by the name of the Agent written in block letters, followed by an underscore and appended with the word 'Program' (e.g. SEM_Program).
- Function names start with a lowercase first letter. The individual words start with block letters and the rest of the letters are written in lowercase (e.g. bestPathPref).
- Abstract rule names are written in block letters and the individual words are divided by an underscore _ (e.g. CHOOSE_NEXT_ROAD).
- Rule names start with a block letter. The individual words also start with block letters and are separated by an underscore _. The rest of the letters are written in lower case (e.g. Choose_Next_Road).
- To denote an inheritance like relation between domains, the specialized domain is written on the left hand side of the $\equiv$ sign, the general domain is written on the right hand side, and this is appended by a keyword '*where*' (e.g ACTIVE_OBJECT $\equiv$ PASSIVE_OBJECT *where*). Following *where* are the functions defined on the specialized domain; these functions may be interspersed throughout the model.
- ASM keywords are written in lowercase using bold font (e.g. **else**).
- Domains are written in block letters (e.g. PERSON).
- Comments inserted within specifications are preceded by '//' and are usually gray colored.

PART II

ABSTRACT MODEL

Chapter 4

Overview of the Model

We model a social system composed of criminals, cops, regular people, and other entities, interacting with each other and their environment as a discrete time, discrete event model, using the ASM formalism.
The goal of the model is to evolve the activity and awareness spaces of people over a period of time. Using the simulated awareness spaces, for a given hypothetical offender, the crime occurrence space is simulated.

The DASM model we present here is built on a multi-agent system view as explained in Chapter 5. The system context calls for a combination of two basically different views — agents interacting with each other and with the environment in which they live. These interactions form a dynamic open system, whose behavioral pattern is best characterized as a combination of *reactive system* and *cognitive system.*

We perform *Micro Simulation*, whereby we model the behavior of constituent individual units i.e the agents, from which the aggregate behavior (macro-level) of the populations, or groups of agents, can be obtained.

Persons are viewed as autonomous agents with a complex architecture, based on a BDI-theoretic view (Chapter 7).
Agents operate in a highly dynamic, discrete environment. The environment is typically an urban landscape; the logical representation of the environment is categorized into different layers to derive the true behavior of the agents, as explained in Chapter 6.

To model the interaction between an agent and its environment, we use the notion of *open system view*. The boundary between the system and its operational environment is delineated using well-defined ASM functions that allow one to model two-way interactions. The system is embedded into its environment through actions and events as observable at interfaces. The external world affects the system though externally controlled or *monitored* functions.
Timing aspects are modeled based on an abstract notion of global system time. In a given state, the time (as measured by some global clock) is given by a monitored unary function *now* taking values in a linearly ordered domain TIME.

The decision-making process of autonomous agents is emulated using a combination of *case-based reasoning* and *model-based reasoning*. The idea is to have a balance of *learning* and *exploring*; this approach closely resembles intuitive human reasoning. Coupled with the process of reasoning thus is the process of learning. The autonomous agents are capable of learning using a form of *behavioral reinforcement learning*, where based on past experiences, certain preferences are developed that may influence future choices that an agent will make. The learning reinforcement can be both positive or negative in effect. This hybrid case-based reasoning is explained in Chapter 8.

Inter-agent communication is achieved by reading and writing shared locations of ASM global states. For synchronization purposes, we use an explicit event triggering mechanism defined in [41].
Intuitively, communicating agents may develop various forms of interactions like *indifference*, *cooperation* or *antagonism*, which in turn lead to the formation of organizations such as egalitarian or hierarchical [44]. At this stage, we essentially restrict to two forms of interaction, indifference (agents operate independently without influencing one another) and, to some extent, cooperation, but potentially allow for other forms of interactions as well. This type of interaction renders an organization structure that is typically egalitarian, however with future inclusion of complex interactions, the structure may potentially emerge into a hierarchy, community of experts, market, scientific community [89].

The behavior of agents is simulated by making them follow their daily routines, and carry on with day-to-day activities. In due course of time, with the movement of people in the given urban environment, their *activity space* and *awareness space* starts building and expanding. People thus start exhibiting relatively fixed temporal and spatial patterns of movement. We

use the *Brantingham Model of Crime Selection*, as explained in Chapter 2, to get an estimate of the crime selection sites. For criminally-disposed people, given a predefined *opportunity space*, the intersection of their awareness space with the opportunity space gives us the *crime occurrence space*. This area has a very high probability of criminal activities. Within this area, based on their criminal profile and characteristics of the target, the criminals then perform *target templating* which leads to victimization of targets.

Chapter 5

Modeling Paradigm

Traditionally, statistical models, differential equations, and stochastic modeling have been the dominant approaches for modeling discrete event systems [95]. However, for modeling emergent human behavior amidst dynamic environment — a particular kind of discrete event system — multi-agent systems prove to be a very promising approach.
This chapter is devoted to examining the suitability of multi-agent systems as a modeling paradigm in the context of our application. We begin with Section 5.1 providing a brief overview of the eclectic field of multi-agent based modeling; this is continued in Section 5.1.1 that analyzes their applicability to modeling of social processes in particular, and Section 5.1.2 that explores the need for introducing formal approaches to agent based systems. Finally, Section 5.2 explains our approach, and how the ASM formalism is combined with the multi-agent system view to obtain a robust modeling paradigm for social systems.

5.1 Multi-Agent Based Modeling

The field of MAS emerged from the study of Distributed Artificial Intelligence (DAI) in the 1980s. The research areas of multi-agent systems and distributed systems coincide, and form the research area of distributed agent computing; in short, multi-agent systems are often distributed systems, and distributed systems are platforms to support multi-agent systems. Multi Agent Systems (MAS) are electronic or computing models made up of artificial entities which communicate with each other and act in an environment [44]. The study of MAS

focuses on systems in which many intelligent agents interact with each other. The agents are considered to be autonomous entities, such as software programs or robots; their interactions can be either cooperative or selfish.

Multi-Agent Systems (MAS) [89], [92], [44] have gained widespread popularity as a modeling paradigm due to their inherent ability to model a plethora of systems — comprising of simple entities to groups of complex entities, emanating simple to complex interactions, embedded in static to open dynamic environments[1].
Multi-Agent Based Systems (MABS) should not be seen as a completely new and original modeling and simulation paradigm. It is influenced by and partially builds upon some existing paradigms, such as, parallel and distributed discrete event simulation, object oriented simulation, as well as dynamic micro simulation [31].

The motivation behind using MAS as a modeling platform are many-fold: to solve problems that are beyond the realm of a single entity/agent due to bounded rationality, to provide solutions in situations where expertise/information sources are distributed e.g. weather forecast system, to provide modularity, to model open and highly dynamic environments [89]. In particular, agent-based models have enormous potential to capture the dynamics of systems that can naturally be regarded as a society of interacting agents, e.g. human societies [52], [3].

Pertinent to our context, the multi-agent modeling paradigm plays a vital role in bringing together the predominant view of the world of social systems with the formal ASM view. They serve as a natural ontology for modeling social phenomenon [3]. This nexus provides a coherent framework to incrementally refine the model by integrating approaches from other disciplines such as AI, algorithmics, criminology, etc and thus incorporating cross-disciplinary perspectives in an elegant fashion. Section 5.1.1 explores the suitability of multi-agent systems for modeling social systems.
As a secondary benefit, the use of ASM formalism in specifying such a system helps understand and alleviate many of the semantic and methodological problems met in the multi-agent system community. This is discussed in Section 5.1.2.

We now recall some basic definitions of an agent and a multi-agent system. There is a bewildering list of notions of agents [48]; however, there is no universally accepted definition.

[1]See the Springer series on *Multi-Agent Systems and Application* for innovative applications of MAS.

Furthermore, a clear distinction exists between Agents and Autonomous Agents that cannot always be explicitly defined. Ferber [44] describes an agent as a "*physical or virtual entity which is capable of acting in an environment, which can communicate, which is driven by a set of tendencies, which possesses resources of its own, which is capable of perceiving its environment,which has only a partial representation of this environment, which possesses skill....*". In [48] the authors describe the '*essence of agency*' and offer the following definition: "*An autonomous agent is a system situated within a part of an environment that senses environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.*" In [91], the agent definition reads as: a hardware or a software-based computer system that enjoys the following properties: *autonomy:* agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state; *social ability:* agents interact with other agents via some kind of agent-communication language; *reactivity:* agents perceive their environment, and respond in a timely fashion to changes that occur in it; *pro-activeness:* agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative.

The term multi-agent system is applied to a system comprising of (1) an environment (2) a set of objects (3) an assembly of agents (4) relations which link objects and agents to each other (5) operations of agents to act on objects, and (6) laws of the universe [44]. The characteristics of MAS are that (1) each agent has incomplete information or capabilities for solving the problem and, thus, has limited viewpoint (2) there is no system global control, and (3) data are decentralized, and computation is asynchronous [89].

Although MABS enjoys a lot of benefits, the framework, methodology and software engineering techniques for same are not well-established. In [38], the authors define the success of MABS as ambiguous: *"While most of the researchers seem to agree on a common terminology for designating the core multi-agent concepts used in MABS, it appears that this agreement is, at best, syntactic. The semantics associated differ considerably from one model to another, or from one implementation to another. This fuzziness, at the computational level...can be found in all the other levels required for the design of a simulation."*

5.1.1 Agent Based Social Simulation (ABSS)

A social system can be seen as a set of actors, individuals or groups behaving in an interdependent pattern in order to adapt to environmental contingencies [52]. The authors in [53] trace the history of simulation in the field of Social Sciences – from its' genesis in differential equations to stochastic processes, to game theory, to cellular automata and finally blossoming to distributed artificial intelligence. They compare the above modeling techniques and extol the usefulness of agent-based modeling to capture non-linear system dynamics. Bankes [3] terms the use of agent-based modeling in the social sciences as revolutionary and extols its virtuosity in meeting the challenges demanded in the social sciences, that differential equations and statistical models cannot meet. Agent-based models have enormous potential to capture the dynamics of systems that can naturally be regarded as a society of interacting agents, e.g. human societies; they serve as a natural ontology for modeling social phenomenon [3] [52].

One way of characterizing the inter-disciplinary research area of Agent-Based Social Simulation (ABSS) is that it constitutes the intersection of three scientific fields - Agent-Based Computing, the Social Sciences, and Computer Simulation [31] (Figure: 5.1).

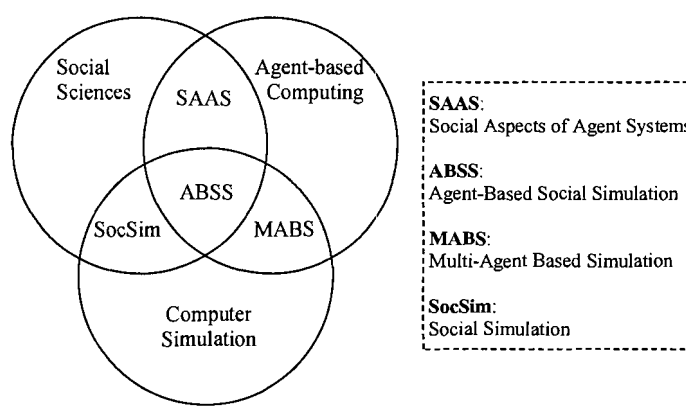

Figure 5.1: *ABSS as the Intersection of Three Fields.*

There are numerous advantages in using agent-based models of social phenomenon. Drougal and Ferber [37] cite that they allow one to *Test Hypotheses* about the emergence of social structure from individual behaviors, i.e by experimenting at the micro level and deriving

patters at the macro-level; *Build Theories* that contribute to sociological development, by relating behavior to structure; *Integrate Differential Partial Theories* from different disciplines into a general framework, by providing tools that allow for integration.
They allow one to capture complex interdependencies among a large number of units, incorporate interdisciplinary perspectives, follow reasoning by simulation runs, handle large amounts of data [52]. Furthermore, mathematical models of social processes provide rigorous a priori frameworks that allow formal or informal reasoning of the target system [84].

Gilbert and Doran [52] cite two main categories of agent-based social models: (1) *Exploratory* models: that give insights into systems (2) *Predictive Models*: to predict reliably the behavior of the target system under key conditions. They also state two kinds of inferences that can be draw on the model: (1) By *Simulation*: having a running executable version of the model and performing experiments on it (2) By *Analysis*: reasoning directly from the knowledge embedded in the model. The reasoning can be informal or formal and is usually applied on the formal specification of the model.

There are two kinds of Social Simulation viz. *Micro Simulation*, that explicitly attempts to model the behavior of constituent individual units, from which the aggregate behavior is obtained and *Macro Simulation* which models the system as a whole where the characteristics of a population are averaged together. The two classifications of simulation are not completely orthogonal, for example in order to aggregate behavior, we might want to give an agent a model of other agents in the environment.

Many successful applications of agent-based social simulation can be found in [52]. The MANTA system [37] and the EOS project [36] are two well-known examples of a reactive and a cognitive system respectively. MANTA uses a multi-agent model to show the evolution of ant colonies; EOS simulates the process of emergence of fishermen societies.

5.1.2 Formal Approaches to Agent-Based Systems (FAABS)

Although agent based systems are widely used, they still lack a formal semantic foundation. Such a firm foundation is quintessential to the process of good design and quality construction of agent-based systems. Introducing formal methods in the realm of agent-oriented analysis and design may serve as a solution to many problems being met in the MAS research.

The authors in [34] contend that there is a dire need for formalism in agent-based systems: *"There is a lot of formal theory in the area but it is often not obvious what such theories should represent and what role the theory is intended to play. Theories of agents are often abstract and obtuse and not related to concrete computational models."*
The same theme is reinstated in [35] - *"Our own view is that work on formal models of agent-based systems are valuable inasmuch as they contribute to a fundamental goal of computing of building real agent systems."*
Furthermore, the lack of a mature off-the-shelf methodology that provides a practical framework guiding the process of specification, design, development and verification inhibits their applicability to real agent commercial applications [61] [85].

Invernoe et al [34] point out two directions in which this field can develop viz: construct new techniques for reasoning about and specifying multi-agent system or use existing formalisms as far as possible.
Wooldridge and Fisher [46] outline a formal approach for the specification, verification, and rapid prototyping of multi-agent systems. The agent specification is developed in a temporal multi-modal belief logic called PML. For prototyping, the agent-specifications are made executable by using the 'Concurrent METAEM' platform. Verification is done by using an extended temporal belief logic ; a range of proof methods for TBL are also developed. Limitations include inability to address true concurrency and distribution, inability to directly execute logic specifications, non-applicability of TBL to realistic systems, complicated proof-methods.
Luck and Inverno [71] propose a formal framework for *Agency and Autonomy.* Using the Z specification language, they describe a three-tiered hierarchy composed of *Objects,Agents, and Autonomous Agents.* They extend their work in [35] by addressing methodological issues of agent systems specification, agent development and agent deployment. The concepts of 'inter-agent relationships', 'sociological behavior' and 'agent plans' are also formalized. The critique of their work include the limitations of the Z specification language [45]. They also do not address certain key issues of agent memory, external environment, goal adoption etc [90].
Hilaire et al [61] present a formal approach to MAS that fits in with prototyping and simulation. They choose a multi-formalism approach based on Object Z and state charts. MAS is specified based on an organizational model which has three interrelated concepts: *Role,*

Interaction and Organization. Object-Z is used to specify the transformational aspects of the system and state charts to specify the reactive aspects.
There also exist a number of agent-based modeling techniques and platforms such as DESIRE, SWARM, REPAST etc. [51] gives a brief sketch of the history of these platforms for modeling agent-based systems. A comparative study of some of these techniques is carried out in [85]. The conclusion drawn in both the cited works is that there is a need for further exploration of agent-based modeling techniques.

Based on the above rhetoric, one can draw the conclusion that relatively little work has been done in terms of formal approaches to agent-based systems. Researchers have enumerated various facets that an agent formalism should address. Main amongst these are *agent-based characteristics* such as distribution, concurrency, autonomy, communication; and *software engineering characteristics* such as preciseness, refinability, executability, methodology [52] [85] [34].
To this end, our work based on the ASM formalism provides a mathematical framework to alleviate and solve the aforementioned crucial problems. It fills the dire need for a robust formal framework that, beyond issues of formalization, also deals with methodological aspects and software engineering techniques. ASM formalism and abstraction principles in combination with the underlying methodological framework [6] provide a universal formal basis for semantic modeling of multi-agent systems at arbitrary levels of abstraction in a coherent and consistent framework. Specifically, we address here crucial aspects such as distribution, concurrency, communication, environment, and real time [85].

5.2 Our Approach: Linking Social Systems to DASM Models

The DASM model we present here is built on a multi-agent system view. We build upon the existing knowledge and expertise of applying the multi-agent based paradigm to modeling of social systems, and then use the DASM framework for building a concise mathematical model of a particular class of such systems. This is depicted in Figure 5.2, where the derived DASM model is linked to the underlying social system through an intermediate layer which is of Multi-Agent System (MAS).

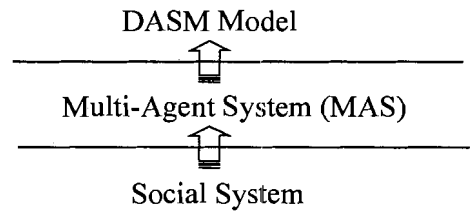

Figure 5.2: *Mapping Social Systems to DASM Models*

The primary reason for the mapping is that this is the intuitive way to model the system; fundamentally the system is nothing but a set of agents interacting with each other in the given environment. Agent-based models have enormous potential to capture the dynamics of systems that can naturally be regarded as a society of interacting agents, e.g. human societies; they serve as a natural ontology for modeling social phenomenon [3] [52].

" Agent-based modeling is relatively new to the social science, but holds the promise of becoming a powerful new computational tool in crime analysis and in policy analysis. With agent-based modeling it is possible to see that what happens to be very different under different conditions...different urban backcloths, but with the agents acting under the same rules; or the result of agents acting under the same rules but against different urban backcloths [17]."

Thus, in our model, the multi-agent system layer acts as a linchpin; it renders an organization to the underlying social system and makes it amenable for the task of formalization. It also provides an avenue for incorporating various techniques and solutions developed in

the discipline of MAS research for common problems encountered in the design of agent-based systems, such as coordination & planning, conflict-resolution, team formation etc [89]. Although at this level of abstraction the inter-agent interaction is minimal, and the above techniques are typically not required, we nevertheless provide a basis for future extensions of the model in this manner.

At the same time, the reader should be cognizant of that fact that we view the introduction of this intermediate layer only as a facilitating means for modeling social systems; i.e. only those principles of multi-agent systems that help in streamlining our target system are incorporated. This stems in part from the fact that although the discipline of MAS is well-established, there is still no universally accepted methodology for engineering agent-based systems, and often the designer is at discretion to make application-specific design choices.

Having explained the general idea of deriving a DASM model of the social system via the multi-agent system layer, in the following section (5.2.1) we explain the specifics that need to be achieved in order to do so.

5.2.1 Classification of Agents: DASM Organization

The above approach of juxtaposing the DASM model with the social system via the MABS layer calls for orchestrating first, the constituent entities within each layer and second, the mapping of entities derived in each layer to its immediate above layer. This sections explains how the classification and mapping of entities is done to obtain a solid DASM organization.

At the bottom most layer, the social system is composed of different types of entities. There are independent live entities with self-controlled action and behavior, such as criminals, cops, regular people etc. There are lifeless entities with behavior albeit not self-controlled, such as traffic lights, atms, buses etc. Finally, there are entities with properties but no behavior such as buildings, streets, valuables etc.

Since the above social structure is mapped to a Multi-Agent System (MAS), it is required to derive the same distinction in this intermediate layer as well. This task of classification of agents in the multi-agent system layer also subsumes making a distinction between *agency* and *autonomy*. We propose a generic hierarchical classification of entities into three different categories: *passive objects*, *active objects* and *autonomous agents*. The core of this

classification is based on the essentials of the *Belief Desire Intention* (BDI) agent architecture [23], [91] and the *framework for agency and autonomy* [71].
A *passive object* is an entity that comprises of a set of *attributes*. An attribute is nothing but a characteristic feature; for instance, *jewelery* is a passive object that has attributes such as value, color, shape.
An *active object* is an entity that, in addition to a set of attributes, has an associated *behavior*. Behavior is described as an observable change in the internal state of the entity or its external environment, where this behavior is induced by the environment (or some external entity) and not generated by the active object itself. A *car* is an active object that has a set of attributes (color, model, engine) and changing behavior (running, stationery, honking), where this behavior is controlled by an external entity (the driver).
An *autonomous agent* is an entity that, in addition to attributes and behavior, has a set of *rules*, *motivations*, and a *memory*. The behavior of an autonomous agent is generated by the rules triggered by the agent itself to change its internal state (by way of *cognition rules*) or the state of its environment (by way *action rules*). Consequently, an autonomous agent is responsible for generating all its behavior. Motivations are reasons (goals or incentives) toward which the behavior is oriented, and memory is nothing but a collection of facts representing agent's knowledge of the environment. A *criminal* is an autonomous agent that has attributes (age, race, sex), behavior (working, eating, stealing) which is determined by motivations (hunger, greed), and has a memory that saves the agent's knowledge about the environment such as the locations of the targets. The definition of the autonomous agent is in essence similar to an BDI agent, whereby, analogous to a BDI architecture, memory represents the beliefs, motivations represent the desires, and the rules represent the deliberative and means-end reasoning phase of the BDI agents.
This three-tier hierarchy of entities is depicted in figure 5.3.

It should however be noted that although generic, this categorization is not intended to be a universal taxonomy for classification of sociological agents. It is derived mainly to capture the dynamics of our target system, which is a typical example of a social system, and thus may be carried forward as a generic classification for social systems. This same line of thought is reinstated by Russel and Norvig *"The notion of agent is meant to be a tool for analyzing systems, not an absolute characterization...."*

The last step is to map each MABS entity onto an entity in the DASM model. As explained

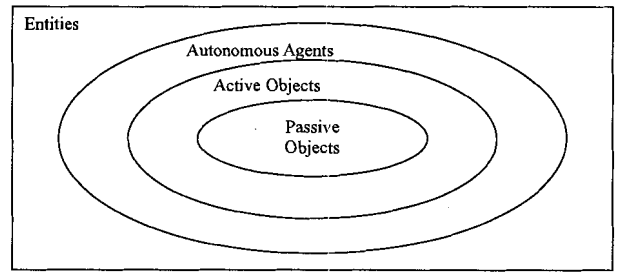

Figure 5.3: *Hierarchical Classification of Entities*

in Chapter 3, there are only two kinds of entities in ASMs viz. *agents* that have an associated program, and *objects* that only have characteristic functions. A passive object is modeled as an ASM object with static functions representing its attributes. Active objects are also modeled as ASM objects with both static and dynamic functions; dynamic functions are used to represent the changes in state corresponding to externally-controlled behavior. Finally, autonomous agents are modeled as DASM agents where the program of each agent characterizes the rules governing its behavior; its memory and motivations are abstractly represented by functions.

Table 5.1 illustrates the entity mapping through the different layers for some typical entities.

Social System	**MABS Model**	**ASM Model**
Offender, Victim	Autonomous Agent (Attributes, Behavior, Rules, Memory, Motivations)	DASM Agent
Car, ATM	Active Object (Attributes, Behavior)	Object (Static/dynamic functns.)
Cash, Drugs	Passive Object (Attributes)	Object (Static functions)

Table 5.1: *Entity Classification and Taxonomy through Different Layers.*

Spec 5.1 shows the ASM snippet of classification and mapping of entities through the different layers [2].

[2]While we use explicit domains here for MEMORY, RULES, BEHAVIOR, and ATTRIBUTES, in the subsequent instantiation of autonomous agents, we don't extend these domains vis-a-vis. Instead, we may use abstract and derived functions, and other domains, that constitute these domains indirectly.

```
// ———————— Entity Classification and Mapping ————————
// Social Entities
domain COP, CRIMINAL
domain CAR, BUS
domain DRUGS, CASH
// MAS Entities
domain ENTITY
domain PASSIVE_OBJECT
domain ACTIVE_OBJECT
domain AUTONOMOUS_AGENT
ENTITY ≡ PASSIVE_OBJECT ∪ ACTIVE_OBJECT ∪ AUTONOMOUS_AGENT
// ASM Entities
domain AGENT

// ———————— Mapping ————————
PASSIVE_OBJECT ≡ DRUGS ∪ CASH
ACTIVE_OBJECT ≡ CAR ∪ BUS
AUTONOMOUS_AGENT ≡ COP ∪ CRIMINAL

AUTONOMOUS_AGENT ≡ AGENT  where

// ———————— Hierarchical Classification ————————
domain ATTRIBUTES, BEHAVIOR, RULES, MEMORY, MOTIVATIONS

// —Passive Object—
static attributes : PASSIVE_OBJECT → ATTRIBUTES

// —Active Object—
ACTIVE_OBJECT ≡ PASSIVE_OBJECT  where
dynamic behavior : ACTIVE_OBJECT → ATTRIBUTES

// —Autonomous Agent—
AUTONOMOUS_AGENT ≡ ACTIVE_OBJECT  where
rules : AUTONOMOUS_AGENT → RULES
  rules(a) ≡ Program(a)
memory : AUTONOMOUS_AGENT → MEMORY
motivations : AUTONOMOUS_AGENT → MOTIVATION – Set
```

Spec 5.1: *Hierarchical Classification and Mapping of Entities.*

Chapter 6

Representation of Environment

The relation between Man and his Environment has been of interest to scholars of diverse disciplines. As highlighted in Chapter 2, one of the major stances taken in Environmental Criminology is that the movements and behavior of people is influenced by the underlying urban landscape, accompanied by their subjective perceptions of the same objective environment. The agents are embedded in and operate within the confines and vagarities of a given environment, which is a dynamically changing reality. This thus calls for a robust representation of the environment in a manner that can include the notions of both *objective reality* and *subjective perceptions*. This chapter explains the approach taken on abstract formal representation of the environment. We begin with Section 6.1 that explores comprehensively the extant views of the criminologists and psychologists; we conclude with Section 6.2 that explains our proposed approach which attempts to complement the popular views.

6.1 Overview

The relation between Man and his Environment has been of interest to scholars of diverse disciplines. Although the disciplines might be different, the views bear close semblance. In this section we comprehensively present the broad outlook on environment held by criminologists and psychologists.

Environmental criminologists set out to use the *geographic imagination* in concert with the *sociological imagination* of people to describe, understand, and control criminal events [11]. This maxim thus encompasses two different views of the environment — an *objective view* and a *subjective view*. The objective view describes the geographic environment as a universal mathematical reality, that includes the totality of all things. This objective space has the same interpretation at all times for all individuals. The subjective view relates to an individual's *perception* of the objective geographic environment. Each individual perceives his or her environment differently which is shaped by a number of factors such as sociological, economical, biological etc. These perceptions are also based on past knowledge and current experiences. For e.g. an infant's understanding and knowledge of its environment is very different from an adult's understanding.
Criminologists contend that absolute geographic arrangements cannot be used in crime analysis without being transformed into subjective coordinates [10].

The criminological thinking is greatly influenced by works of leading psychologists. Immanuel Kant argued that we do not perceive the world as it is; we impose cause and effect relationships on it and therefore our perceptions are influenced by our experiences.

The classical work of Koffka in his book *Principles of Gestalt Psychology* [64] explains the relation between nature and human mind. The author categorizes the environment into *geographical environment*, which is the actual physical structure, and *behavioral environment*, which is an individual's perception of the geographical environment. People behave in the ways they do based on how they perceive the environment (behavioral) instead of how the environment actually is (geographical). The practical application of this would be in understanding someones behavior within the context of their environment instead of our own. Further, he classifies peoples' reactions into two categories, viz. *distal stimuli* and *proximal stimuli*: distal stimuli describes things as they exist in the geographical environment and proximal stimuli are the effects that distal stimuli have on sensory perception.

The same theme runs in [86], where the author in addition to propounding the notion of geographic and subjective environment, categorizes the subjective environment into three sub-categories with a hierarchical relation — *operational environment*, which is that part of the geographic environment that has impact on an individual; *perceptual environment*, that part of the operational environment that an individual is aware of; and *behavioral environment*, that part of the perceptual environment which triggers responses.

6.2 Our Approach: Layering of Environment

The approach we propose on modeling the urban environment attempts to embody and complement the extant theories of environment in the behavioral sciences, as discussed above. The view incorporates the fundamental fact that environment can be divided into two broad categories — the *objective environment* and the *subjective environment.*

The objective environment, in our case called the *Geographic Environment*, is external to an agent and encompasses the totality of all things. This is the physical reality and cannot be manipulated by the agents. All agents have the same representation and interpretation of the geographic environment. It is potentially composed of the natural typography, roads and street network, transit and traffic patterns, people and other objects. Section 6.2.1 discusses the representation of geographic environment formally.

The subjective environment, in our case is called also *Subjective Environment* and is divided into three sub-categories with a hierarchical relation viz. *Perception, Awareness Space and Activity Space.* The subjective environment is specific to an agent (person) as opposed to being universal to all agents.

Perception is how each person 'perceives' the objective environment (geographic environment), depending on personal preferences, past knowledge and current experiences, and other socio-cultural factors. For instance, an ignorant person might perceive Antarctica to be a desert; a person traveling to work might perceive a road with heavy traffic as being unfavorable, whereas a car thief might perceive this road as highly favorable with good opportunities. This perceived environment is the effective environment that triggers agent behavior. A person, in general, cannot possibly 'perceive' the entire geographic environment; *perception* is thus a sub-part of the *geographic environment.* The notion of perception is similar to a mental map, as discussed in Chapter 2.

That sub-part of the *perception*, that an agent is aware of by way of current events, past experiences, interaction with other agents, forms the *Awareness Space* of the agent. It is composed of all locations in the *perception* about which a person has knowledge above a minimum threshold level even without visiting some of them. For instance, the ignorant person hears from someone that Antarctica is not a desert but a polar region, so this information becomes a part of his or her awareness space; while traveling on an edge a person looks around and admires the scenery, this admired scenery then becomes a part of his

awareness space. This notion of awareness space is the same as that of criminologists, as discussed in Chapter 2.

Activity Space is that sub-set of the *awareness space* that the agent has physically traveled on frequently in a given past time interval. The general paths treaded by people during everyday activity, such as home-work, work-lunch, home-recreation, form the activity space. The agent typically has very detailed information about this part of the environment. This notion of Activity Space corresponds to that of the criminologists (Chapter 2).

Section 6.2.2 explains the representation of subjective environment in formal terms.

The aforementioned categorization of environment helps in deriving the true behavior of the agent with respect to criminal activity and is illustrated in Figure 6.2.

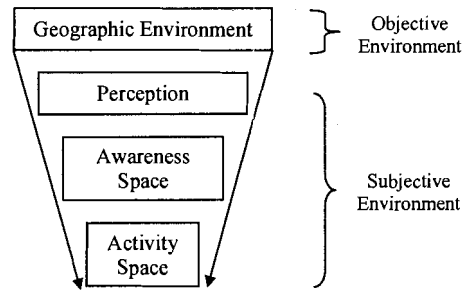

Figure 6.1: *Categorization of Environment*

Further, we base the structuring and representation of the environment using *graph theory*. The following sections explain the formalization of environment discussed in this section, in graph theoretic terms and alongside in ASM terms, in a step-by-step manner.

6.2.1 Objective Environment

The objective environment, in our case called the Geographic Environment, represents the urban landscape. We abstractly represent the given geographic environment, as a *directed attributed graph* defined in several steps as follows.

Let $H = (V, E)$ be a directed graph representing the road map of some urban area. $V = \{v_1, ..., v_n\}$ is the set of vertices [1]; a vertex represents the intersection of two edges on the

[1]We use the terms vertex and node interchangeably.

map. $E = \{e_1, ..., e_n\}$ is the set of directed edges representing roads, where $E \subseteq V \times V$; unidirectional roads are represented by a single edge and bidirectional ones by a pair of oppositely directed edges connecting the same two vertices. This road map is formally defined in Spec 6.1.

```
// ———————— The Environment Graph ————————
// H = ENVIRONMENT_GRAPH, v = NODE, e = EDGE
domain ENVIRONMENT_GRAPH
domain NODE
domain EDGE

// ———————— Graph Structure ————————
// (H = V, E), V = nodeSet, E = edgeSet
nodeSet : ENVIRONMENT_GRAPH → NODE – set
edgeSet : ENVIRONMENT_GRAPH → EDGE – set
```

Spec 6.1: *Representation of the Road Map.*

We can then define some general operations on the *NODES* and *EDGES* of the environment graph (Spec 6.2).

```
// ———————— Node ————————
outIncidentEdges : NODE → EDGE – set
adjacent : NODE × NODE → BOOLEAN

// ———————— Edge ————————
edgeHead : EDGE → NODE
edgeTail : EDGE → NODE
```

Spec 6.2: *Some Operations on Nodes and Edges.*

Next, for attributing the graph H, let $\Theta = (\Theta_v, \Theta_e)$, where Θ_v and Θ_e denote the attribute sets for vertices and edges respectively. Θ_e splits into two disjoint subsets, Θ_e^{stat} and Θ_e^{dyn}, the edge attributes that are statically defined, such as distances, and those that my change dynamically, like traffic conditions, respectively. Similarly, Θ_v splits into two disjoint subsets, Θ_v^{stat} and Θ_v^{dyn}, the node attributes that are statically defined, such as location, and

those that my change dynamically, like density of people. The terms static and dynamic are used to refer to changes with respect to time. Static attributes typically do not change during the course of a simulation run, whereas static attributes represent dynamic fluctuating conditions and may change during simulation runs. Spec 6.3 represents this categorization of geographic attributes.

// ——————— Graph Attributes ———————
domain GEO_STAT_NODE_ATTR // Θ_v^{stat}
domain GEO_DYN_NODE_ATTR // Θ_v^{dyn}
// $\Theta_v = (\Theta_v^{stat}, \Theta_v^{dyn})$
domain GEO_NODE_ATTR $\equiv$ GEO_STAT_NODE_ATTR $\cup$ GEO_DYN_NODE_ATTR

domain GEO_STAT_EDGE_ATTR // Θ_e^{stat}
domain GEO_DYN_EDGE_ATTR // Θ_e^{dyn}
// $\Theta_e = (\Theta_e^{stat}, \Theta_e^{dyn})$
domain GEO_EDGE_ATTR $\equiv$ GEO_STAT_EDGE_ATTR $\cup$ GEO_DYN_EDGE_ATTR

// $\Theta = (\Theta_e, \Theta_v)$
domain GEO_ATTR $\equiv$ GEO_EDGE_ATTR $\cup$ GEO_NODE_ATTR

Spec 6.3: *Categorization of Geographic Environment Attributes*

We can now define the objective environment, henceforth called *Geographic Environment* as an attributed directed graph $G_{GeoEnv} = (H, \theta)$, where θ represents the collective geographic attribution. Formally, this is expressed by an attribution scheme $\theta = (\theta_v, \theta_e)$ with $\theta_e = (\theta_e^{stat}, \theta_e^{dyn})$ representing the mapping of edge attributes, and $\theta_v = (\theta_v^{stat}, \theta_v^{dyn})$ representing the mapping of vertex attributes. This thus consists of four finite mappings:

- $\theta_v^{stat} : V \rightarrow \mathcal{P}(\Theta_v^{stat})$ assigns a set of static vertex attributes to each vertex in V.
- $\theta_v^{dyn} : V \rightarrow \mathcal{P}(\Theta_v^{dyn})$ assigns a set of dynamic vertex attributes to each vertex in V.
- $\theta_e^{stat} : E \rightarrow \mathcal{P}(\Theta_e^{stat})$ assigns a set of static edge attributes to each edge in E.
- $\theta_e^{dyn} : E \rightarrow \mathcal{P}(\Theta_e^{dyn})$ assigns a set of dynamic edge attributes to each edge in E.

Spec 6.4 depicts this attribute mapping of geographic environment. The abstract attribution schema allows one to include as many attributes as desired as per the need of the application.

// ——————— Geographic Environment ———————
// $G_{GeoEnv} = (H, \theta)$
GEO_ENV ≡ ENVIRONMENT_GRAPH *where*
geoAttr: GEO_ENV → GEO_ATTR

// Geo Environment functions
// $\theta_v^{stat} : V \rightarrow \mathcal{P}(\Theta_v^{stat})$
geoStaticNodeAttr: NODE × GEO_ENV × GEO_STAT_NODE_ATTR → VALUE
// $\theta_v^{stat} : V \rightarrow \mathcal{P}(\Theta_v^{dyn})$
geoDynamicNodeAttr: NODE × GEO_ENV × GEO_DYN_NODE_ATTR → VALUE
// $\theta_e^{stat} : E \rightarrow \mathcal{P}(\Theta_e^{stat})$
geoStaticEdgeAttr: EDGE × GEO_ENV × GEO_STAT_EDGE_ATTR → VALUE
// $\theta_e^{dyn} : E \rightarrow \mathcal{P}(\Theta_e^{dyn})$
geoDynamicEdgeAttr: EDGE × GEO_ENV × GEO_DYN_EDGE_ATTR → VALUE

Spec 6.4: *Geographic Environment*

Specific to our application, we refine *Static Vertex Attributes* to include information such as geographic coordinates. *Static Edge Attributes* yield information on distances, and road type. *Dynamic Edge Attributes* store fluctuating information such as road condition, speed limits, traffic situation (Spec 6.5).

// ——————— Geo Static Node Attribute ———————
coordinate: → GEO_STAT_NODE_ATTR
nodeName: → GEO_STAT_NODE_ATTR
// ——————— Geo Static Edge Attribute ———————
distance: → GEO_STAT_EDGE_ATTR
roadType: → GEO_STAT_EDGE_ATTR
edgeName: → GEO_STAT_EDGE_ATTR
// ——————— Geo Dynamic Node Attribute ———————
traffic: → GEO_DYN_EDGE_ATTR
roadCondition: → GEO_DYN_EDGE_ATTR

Spec 6.5: *Refinement of Geographic Environment Attributes*

6.2.2 Subjective Environment

Having defined the objective environment in terms of the Geographic Environment G_{GeoEnv}, we now have a basis for defining the subjective environment. We model the subjective environment by introducing additional attribution on top of G_{GeoEnv}. The fact that, in general, each agent perceives the geographic environment differently implies that distinct agents see different attributions of the same G_{GeoEnv}. Thus, the subjective environment $G_{SubjEnv}$, can be seen as an attributed directed graph with *colored attributes*. Each color refers to the specific interpretation of an individual agent.
We use λ to denote the collective attribute set of the subjective environment, where λ is composed of three dis-joint sets that represent the attributes of the three layers of the subjective environment.

- λ_{PER} identifies the attribute set of the *Perception*, with $\lambda_{PER} = (\lambda_v^{per}, \lambda_e^{per})$, where λ_v^{per} and λ_e^{per} represent the perception edge and vertex attributes respectively.
- λ_{AW} identifies the attribute set of the *Awarenes Space*, with $\lambda_{AW} = (\lambda_v^{aw}, \lambda_e^{aw})$, where λ_v^{aw} and λ_e^{aw} represents the awareness edge and vertex attributes respectively.
- λ_{AC} identifies the attribute set of the *Activity Space*, with $\lambda_{AC} = (\lambda_v^{ac}, \lambda_e^{ac})$, where λ_v^{ac} and λ_e^{ac} represent the activity edge and vertex attributes respectively.

Further, for the purpose of attribute mapping, we split λ into λ_v and λ_e, to denote the collective attribute sets for vertices and edges respectively from the three layers. It should be noted that there is no classification of subjective environment attributes into static and dynamic; this is because none of these attributes are static but are dynamically changing. This categorization of subjective environment attributes is depicted in Spec 6.6.

The subjective environment is then defined abstractly as an attributed directed graph $G_{SubEnv} = (G_{GeoEnv}, \Lambda)$ where $\Lambda = (\Lambda_v, \Lambda_e)$ abstractly represents the *agent specific* attribution of vertices and edges by means of two injective mappings such that

- Λ_v : AGENT $\times V \rightarrow \mathcal{P}(\lambda_v)$, for each agent and each vertex in V, yields a set of subjective node attributes, and
- Λ_e : AGENT $\times E \rightarrow \mathcal{P}(\lambda_e)$, for each agent and each edge in E, yields a set of subjective edge attributes.

// ——————— Subjective Environment Attributes ———————
domain PER_EDGE_ATTR // λ_e^{per}
domain PER_NODE_ATTR // λ_v^{per}
domain PER_ATTR ≡ PER_EDGE_ATTR ∪ PER_NODE_ATTR // $\lambda_e^{per} = (\lambda_v^{per}, \lambda_e^{per})$

domain AW_EDGE_ATTR // λ_e^{aw}
domain AW_NODE_ATTR // λ_v^{aw}
domain AW_ATTR ≡ AW_EDGE_ATTR ∪ AW_NODE_ATTR // $\lambda_{AW} = (\lambda_v^{aw}, \lambda_e^{aw})$

domain AC_EDGE_ATTR // λ_e^{ac}
domain AC_NODE_ATTR // λ_v^{ac}
domain AC_ATTR ≡ AC_EDGE_ATTR ∪ AC_NODE_ATTR // $\lambda_{AC} = (\lambda_v^{ac}, \lambda_e^{ac})$

domain SUBJ_ATTR ≡ PER_ATTR ∪ AW_ATTR ∪ AC_ATTR // $\lambda = (\lambda_v, \lambda_e)$
domain SUBJ_EDGE_ATTR ≡ PER_EDGE_ATTR ∪ AW_EDGE_ATTR ∪ AC_EDGE_ATTR
domain SUBJ_NODE_ATTR ≡ PER_NODE_ATTR ∪ AW_NODE_ATTR ∪ AC_NODE_ATTR

Spec 6.6: *Categorization of Subjective Environment Attributes*

Spec 6.7 depicts this mapping of subjective environment attributes.

Although *subjective environment* is defined as a total function, i.e every vertex and edge of the underlying graph is associated with all the attribute sets of the three layers of the subjective environment, it is not necessarily the case that all attributes have a 'defined' value. We use default or nullary values for non-inclusion of edges and vertices to reflect the fact that only a sub-set of the graph constitutes a particular layer of subjective environment. For instance, in general it is not necessarily the case that an agent perceives the entire geographic environment; an agent has knowledge of only some parts of the geographic environment, so only those edges and attributes that have a define value for perception attributes form the perception of the agent. The same holds for awareness space and activity space; i.e only those edges that have defined values for awareness and activity attributes form the awareness and activity space respectively of the agent.

The first layer of subjective environment, called *perception*, is the agent's subjective interpretation of the geographic environment. Intuitively, this perception can be seen as a filter through which the agent views the geographic environment. For instance, a road with high density of people may be seen as a nuisance to someone traveling to work; alternately it may

// ——————— Subjective Environment ———————
// $G_{SubEnv} = (G_{GeoEnv}, \Lambda)$
SUBJ_ENV ≡ GEO_ENV *where*
subjAttr : PERSON × SUBJ_ENV → SUBJ_ATTR

// Subjective Environment Related Functions
// Λ_e : AGENT × $E \rightarrow \mathcal{P}(\lambda_e)$
subjEdgeAttr : PERSON × EDGE × SUBJ_ENV × SUBJ_EDGE_ATTR → VALUE
// Λ_v : AGENT × $V \rightarrow \mathcal{P}(\lambda_v)$
subjNodeAttr : PERSON × NODE × SUBJ_ENV × SUBJ_NODE_ATTR → VALUE

Spec 6.7: *Subjective Environment*

be seen as a good opportunity to a pick pocket. We refine λ_v^{per} and λ_e^{per} to include node and edge *Perception Attributes* respectively. The node perception attributes are the same as geographic node attributes, and edge perception attributes are the same as geographic edge attributes. Abstract functions are used to yield the agent-specific subjective interpreted values of the geographic attributes. An agent's *perception* can then be computed by using a derived function on $G_{SubjEnv}$ that extracts the subset of edges with a defined value of perception attributes. (Spec 6.8).

// ——————— Perception Attributes ———————
PER_EDGE_ATTR ≡ GEO_EDGE_ATTR // λ_e^{per}
PER_NODE_ATTRRIBUTE ≡ GEO_NODE_ATTR // λ_v^{per}

// derived function for perception space
perception : PERSON × SUBJ_ENV → EDGE − *Set*

Spec 6.8: *Refinement of Perception Attributes*

The second layer of subjective environment, called the *Awareness Space*, is identified by the attributes λ_v^{aw} and λ_e^{aw} associated with each vertex and edge respectively. The awareness space is a sub-part of the perception, and so only those edges and vertices that have a defined value for λ_{per} constitute the awareness space. We refine λ_e^{aw} to include the attribute *intensity*, which stores the magnitude of the agent's awareness of this edge. The intensity

of awareness depends on a number of factors such as strength of activity space, mobile visibility, accessibility from activity space, willingness to leave activity space, attractiveness of adjacent areas, speed and model of travel etc. We define the intensity of an edge using abstract functions[2]. By means of a *derived function* on subjective environment, we can compute the set of edges that have a defined value for awareness attributes, and thus form the awareness space of the agent. Further, we can impose an additional restriction and return only those edges that have an intensity value above a certain predefined threshold; this is called the *active awareness space* of the agent (Spec 6.9).

```
// ———————— Awareness Attributes ————————
AW_EDGE_ATTR : → intensity

// derived function for awareness space
awarenessSpace : PERSON × SUBJ_ENV → EDGE − Set
activeAwarenessSpace : PERSON × SUBJ_ENV × VALUE → EDGE − Set
```

Spec 6.9: *Refinement of Awareness Space Attributes*

The third and the final layer of the subjective environment, the *Activity Space*, is defined by the attribute set λ_e^{ac}, which stores the values for *frequency, trip importance and reinforcement. Frequency* refers to the number of times an agent has traversed this edge; the more an agent traverses an edge, the more he remembers it for future traversal. *Trip Importance* refers to the significance of the trip on which this edge was taken; edges on important trips (pathway to work) yield a higher value for awareness space and are remembered for a longer period of time. *Reinforcement* refers to the experience an agent had on this edge, edges with a positive (pleasant) experience have a higher likelihood of being taken again, whereas edges with a negative (bad) experience are less likely to be taken again. This enables us to incorporate *reinforcement learning* by defining the reinforcement values of edges abstractly. An agent's activity space can then be computed by using a derived function on $G_{SubjEnv}$ that extracts the subset of edges with a defined value of activity attributes. We can also use a derived function that returns the *active activity space*, which is those set of edges with a compound value of activity attributes that are above a certain threshold (Spec 6.10).

[2]In the simplest case, the value of intensity of awareness space is typically a distance decay function that also considers the strength of the activity space.

// ——————— Activity Attributes ———————
frequency :→ AC_EDGE_ATTR
tripImportance :→ AC_EDGE_ATTR
reinforcement :→ AC_EDGE_ATTR
// derived function for activity space
activitySpace : PERSON × SUBJ_ENV → EDGE − *Set*
activeActivitySpace : PERSON × SUBJ_ENV × VALUE → EDGE − *Set*

Spec 6.10: *Refinement of Activity Space Attributes*

Lastly, the abstract domain VALUE denoting the possible values of the different attributes is refined in Spec 6.11. Detailed listing can be found in Appendix A.

VALUE ≡ REINFORCEMENT ∪ TRIP_IMPORTANCE ∪ FREQUENCY ∪ TRAFFIC
INTENSITY ∪ DISTANCE ∪ ROAD_TYPE ∪ ROAD_CONDITION
DISTANCE ≡ INTENSITY ≡ FREQUENCY ≡ FLOAT

positive :→ REINFORCEMENT
negative :→ REINFORCEMENT
neutral :→ REINFORCEMENT

obligatory :→ TRIP_IMPORTANCE
required :→ TRIP_IMPORTANCE
notRequired :→ TRIP_IMPORTANCE

low :→ TRAFFIC
medium :→ TRAFFIC
high :→ TRAFFIC

minor :→ ROAD_TYPE
major :→ ROAD_TYPE

Spec 6.11: *Refining the Abstract Domain VALUE*

Chapter 7

High-Level DASM Model

This chapter explains the operational DASM model at the highest level of abstraction. Section 7.1 explains the devised architecture of the autonomously acting agents in the model, and describes the functional decomposition of the agents into respective modules. Next, we start detailing the working of the constituent modules of the agents; Section 7.2 describes the *Space Evolution Module*, Section 7.3 illustrates the *Target Solution Module*, and finally Section 7.4 explains the functioning of the *Agent Decision Module*.

7.1 Agent Architecture

This Section explains the architecture of the autonomously acting entity in our system — the Person Agent. The person agent architecture is BDI-theoretic; a brief introduction to the Belief Desire Intention (BDI) architecture is provided in Section 7.1.1, and Section 7.1.2 explains how our approach incorporates this view.

7.1.1 BDI Agent Architecture: Introduction

An agent is characterized by its architecture. The agent architecture dictates how the functional complexity of the agent is organized; the structuring, type of approach for reasoning, the behavior etc. Broadly, agent architectures can be characterized into four categories, viz: *Logic-Based, Reactive, BDI, and Layered* [91].

The *Belief Desire Intention (BDI)* [22] agent architecture, first issued in works of Bratman et al, is the most mature approach for modeling autonomous agents. It has its their roots in the philosophical tradition of understanding *practical reasoning*; deciding what goals we want to achieve, and how we are going to achieve these goals. The former process is known as *deliberation*, the latter as *means-ends reasoning.* The central constructs in the BDI model are *Beliefs*, *Desires* and *Intentions.*

Beliefs represent the *informational state* of the agent, that is, what it knows about itself and the world, by way of past experiences or current events. *Desires* or *goals* are its *motivational state*, that is, what the agent is trying to achieve. Since all desires or goals may not be consistent, the agent has to *commit* to a consistent set of goals and focus actions for these selected goals. These persistent goals are called *Intentions* and represent the deliberative state of the agent, that is, which plans the agent has chosen for eventual execution. To execute intentions, the agent has *procedural knowledge* constituted by a set of *Plans* which are *sequences of actions* to be performed to achieve a certain goal or react to a specific situation. Some view intentions as being a subset of goals which the agent commits to, while others view intentions as the set of selected plans.

Simply stated, *Beliefs* are agents' knowledge of its world; by applying the process of *deliberation* on current beliefs and current intentions, we obtain *Desires*; by committing to desires (also considering beliefs, previous intentions), we obtain *Intentions*; and by applying *means-end reasoning* on beliefs, intentions, desires, we acquire a *Plan of Action*, which upon execution fulfills intentions and eventually desires.

Intentions are fulfilled by executing an action (of the plan) one step at a time. A step can change the beliefs, perform actions on the external world, submit new goals, create new intentions. Since the state of the external world and the internal memory of the agent is continuously changing, it becomes necessary to re-evaluate plans of actions, and reconsider committed intentions, to determine whether one is still on the right track. *Bold* agents never reconsider alternative options, while *cautious* agents constantly reconsider alternative options; a good balance between the two is desirable [50].

The main advantage of the BDI architecture is that it is intuitive, and that it provides a clear functional decomposition of the agent behavior. It has a wide array of applications in both industrial and research applications.

However, there is also rampant criticism of the architecture. The main difficulty is knowing

how to efficiently implement its functionality, since the BDI approach is at a high-level of design. It can be argued that the current BDI architecture lacks an adequate computational model expressing concurrency and distribution semantics, does not clearly show the activities an agent is performing at a given time, and how they relate to one another [77]. BDI model is also inappropriate for building systems that learn and adapt their behavior, and does not provide explicit considerations for multi-agent aspects of behavior [50].
Therefore, in designing our Person Agent architecture, we keep the above limitations in mind, and start with a BDI-theoretic core and build upon it, instead of limiting to a pure BDI architecture. This is explained in the following section.

7.1.2 Our Approach: Person Agent Architecture

We developed our Person Agent architecture as a pragmatic BDI-based model.We start with a BDI-theoretic core, followed by incremental refinements, and incorporate other problem-solving techniques as demanded by the application domain. Having a BDI-theoretic kernel allows us to reap the benefits of the strong theoretical foundation of the BDI model, while we also garner the pragmatics of practical and extensible reasoning techniques, in particular Case-Based Reasoning and Model-Based Reasoning. Our concrete BDI-based architecture can be seen as a system of concurrent sub-systems, that renders the original abstract architecture a firm computational semantics.
Figure 7.1 illustrates the structural decomposition of the person agent into different logical components. It shows the architecture from a functional perspective, which is in tandem with the philosophical foundations of the BDI model.

Memory is a collection of facts/beliefs/knowledge, present and past, that the agent holds about its environment and about itself, which may change dynamically over time. Memory thus represents everything that the agent 'knows', by way of past experiences and current events. The agent's main memory component is the *Subjective Environment* — a filtered view of the geographic environment which is the environment as the agent 'perceives' it; it is thus the subjective reality and is constantly updated by the SEM and the TSM.

The Space Evolution Module (SEM) is responsible for carving out the *activity space* and the *awareness space* of the person agent. It is represented as an ASM agent. It uses a navigation algorithm to move the agent from a given origin to a given destination, considering the

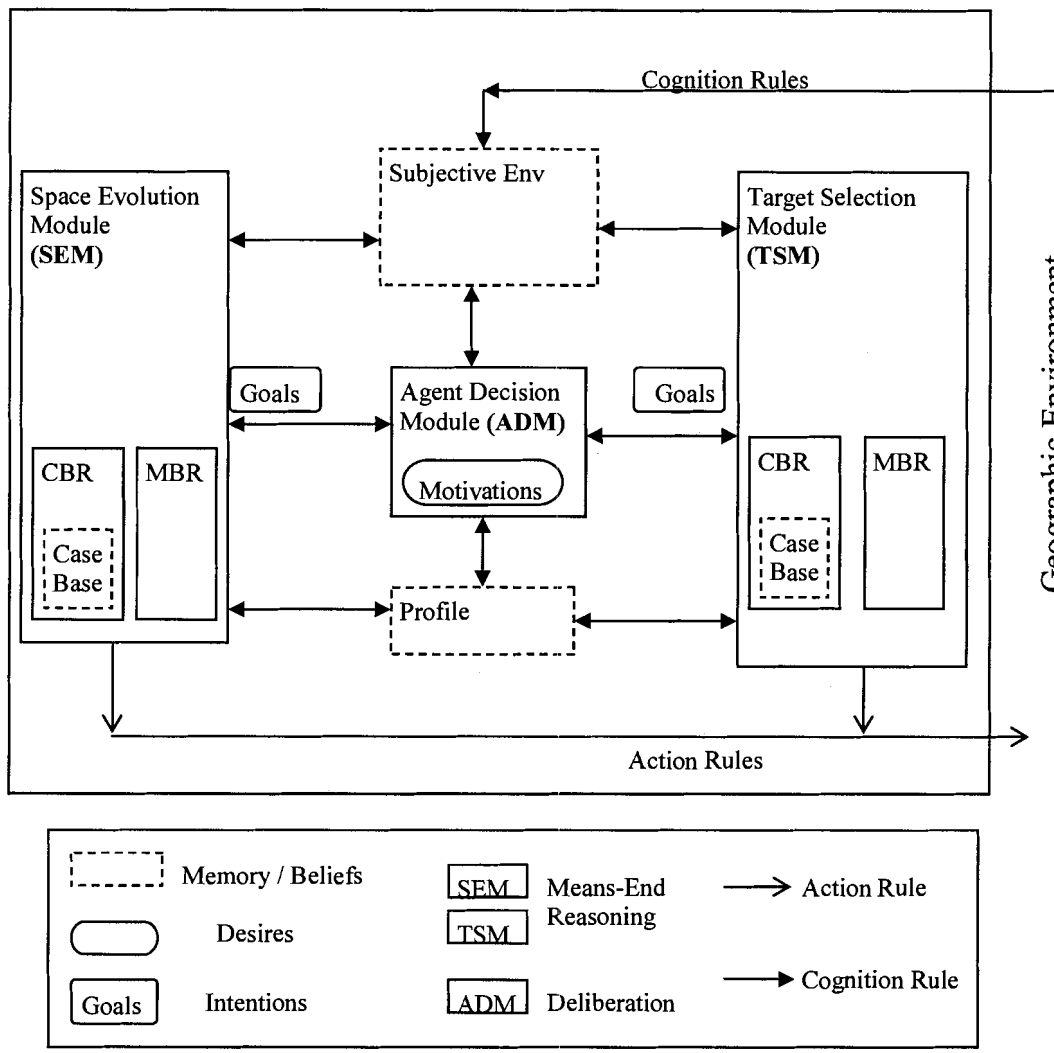

Figure 7.1: *Person Agent Architecture*

particular preferences of the agent. The navigation algorithm uses a 'reasoning mechanism' which combines case-based reasoning (CBR) and model-based reasoning (MBR). The CBR component is represented as an ASM Agent whereas the MBR component can be realized by a submachine. The CBR component works with a *case-base*, which is a type of memory, that stores all previous paths (cases) the agent has taken; i.e an agent remembers previous path planning decisions and merely recollects these decisions in solving a new path planning problem. The MBR component on the other hand uses an algorithmic approach to explicitly solve the path planning problem; it calculates path preferences by considering a variety of aspects such as choosing shorter routes, avoiding traffic, using familiar roads, et cetera.

The Target Selection Module (TSM), implemented as an ASM Agent, is responsible for monitoring potential targets on the routes taken by the agent, and for selecting attractive targets based on given selection criteria. This leads to the creation of the *crime occurrence space* of the agent. For the process of target selection, a hybrid reasoning mechanism composed of case-based reasoning (CBR) and model-based reasoning (MBR) is used; the two components are represented as ASM agents and submachine respectively. The case-based reasoner has a *case-base*, which is a kind of memory, storing previous target selection decisions an agents has made; in selecting a new target, an agent then merely matches the characteristics of the potential target against victimized targets and decides whether it is a 'good' or a 'bad' target. The MBR component on the other hand uses an algorithmic approach to explicitly determine the suitability of the targets.

The Agent Decision Module (ADM) monitors the working of the TSM and SEM and provides relevant inputs to the two modules. It decides on 'what to do' and then relegates the decision to the TSM or the ADM on 'how to do it'. For e.g, it decides that the agent should go from home to work, and then gives this 'goal' to the SEM. The decisions are based on agent's motivations, the current state of the agent, and the information in the memory. Motivations are long-term goals (earn livelihood, greed), that in turn give rise to short term goals (go from home-to-work, car theft), which are then passed to one of the two modules for realization. In our case, motivations are always persistent, whereas goals are non-persistent and change with varying conditions.

The working of the SEM, the TSM, and the ADM is stated using *rules*; intuitively divided into *action rules* and *cognition rules*. Cognition rules change the state of the internal memory, whereas action rules, when fired, affect the state of the external environment.

Finally, the *profile* represents the personal attributes of the agent and stores information such as preferences, skills, home locations, demographic factors etc. The profile of the agent determines crucial aspects of an agent's individual behavior such as moving in space and time, target selection etc.

Based on the above description, it is easy to see the semblance between the pure BDI-model and our concrete BDI-theoretic agent architecture. *Beliefs* are nothing but person agent's memory (subjective environment, profile, case-bases). Motivations are similar to *Desires*, that are derived from beliefs about the world. The ADM represents the *deliberation process* of the BDI, that based on memory, motivations, and current intentions, decides which motivations (in the form of goals) persist. These goals are then communicated to the TSM/SEM, that represent the *Intentions*[1]. The TSM and SEM realize these intentions by generating and executing a plan of action for them; they thus represent the *means-end reasoning* phase of the BDI-model. The SEM and TSM continually monitor the external environment and the proposed plan of action to determine if it needs to be revised, and if so generate an alternate plan; e.g in going from home-to-work, the SEM first proposes a *suggested path*, which after a while might have a heavy traffic jam, in which case the SEM takes this into consideration and proposes an alternate *suggested path*. The ADM too continually monitors the environment, motivations, and goals and revises the motivations and goals if required. For e.g while going to work, a person gets a flat tyre, the person's goal is then be to go to a mechanic instead of work. This new goal (intention) is then passed to the SEM. Further, the cases stored in the case-bases represent the *cached plans*.

Spec 7.1 gives the formal ASM representation of the functional components of the Person Agent at the highest level of abstraction.

In the next level of refinement, we show the constituent CBR components of the TSM and SEM (Spec 7.2). The domain CBR represented as an ASM Agent denotes the Abstract CBR. Instantiations of this abstract CBR yield problem-specific CBRs, TSM_CBR and SEM_CBR, for the modules TSM and SEM respectively. The MBR components are implemented as submachines whose definitions are implicit to the module definitions, and not shown here.

[1]We do not explicitly represent the motivations and goals, but certain abstract and derived data structures represent these implicitly.

```
// ———————— Person Agent Architecture ————————
domain PERSON
// Modules
domain SEM
domain TSM
domain ADM
MODULE ≡ SEM ∪ TSM ∪ ADM
AGENT ≡ PERSON ∪ MODULE  // DASM Agent
// Functions on Person
spaceModule : PERSON → SEM
targetModule : PERSON → TSM
decisionModule : PERSON → ADM
```

Spec 7.1: *Person Agent Architecture.*

```
// ———————— CBR Components of the Modules. ————————
domain CBR
domain SEM_CBR, domain TSM_CBR
CBR ≡ SEM_CBR ∪ TSM_CBR
// Respective CBR components
semCBR : SEM → SEM_CBR
tsmCBR : TSM → TSM_CBR
```

Spec 7.2: *CBR Components of Person Agent Modules.*

7.2 Space Evolution Module (SEM)

The Space Evolution Module (SEM) of the Person Agent is responsible for deciding the path a person chooses from a given source to a given destination. The source and destination are provided to the SEM by the Agent Decision Module (ADM). The problem of finding a path is essentially the problem of navigation, and we can say that the SEM implements a Cognitive Navigation algorithm.

We present here a cognitive navigation algorithm, that integrates learning, exploration and human cognition of geography. The navigation algorithm reflects natural and intuitive choices a person makes while moving in an urban landscape. The path taken might not be a globally optimal and best one, but a more natural and good-enough one. The algorithm takes into account the factors that are known to influence human path planning and is developed in collaboration with the domain experts. Overall, navigation is modeled as a combination of *exploration* and *reinforcement learning*. Reinforcement Learning is achieved by developing a Case-Based Reasoner, which basically implies the person makes all decision based solely on past experiences that are stored in memory. Exploration is achieved by a 'shortest path algorithm', that forms the model-based component of the SEM, which basically algorithmically decides the path based on a number of *influence factors* such as minimizing distance, road traffic; maximizing experience, familiarity etc. Each influence factor is associated with a *weight* that decides its importance in the overall path preference. Human cognition of geography is achieved by basing all decisions considering the person-specific *subjective environment* along with the universal *geographic environment*, as described in Chapter 6.

First, we focus on the high-level view of the navigation algorithm. We formalize the navigation algorithm at the highest natural level of abstraction (level 0). We then apply intermediate refinement steps explained in the subsequent sections, that increase the level of detail in a step-by-step manner (level 1-2). The case-based and model-based components that are derived as a final refinement step of the SEM, form the model at the next major level of abstraction called *Refined Model*, and are discussed in detail in Chapters 8 and 9 respectively.

7.2.1 Level 0

Level 0 of the SEM represents the highest level of abstraction, and describes the SEM and hence the navigation algorithm at the most natural level of detail. To formalize the navigation algorithm as given by the SEM, we first associate certain data structures with this module. Spec 7.3 enumerates the signals that the SEM uses interspersed through all levels of abstraction, and Spec 7.4 gives a self-explanatory enumeration of the constituent data structures of Level 0.

```
// ———————— Signals of SEM ————————
domain INFORM_ARRIVAL // Triggered by SEM to ADM stating destination reached
domain NEW_PROBLEM // Triggered by SEM to CBR stating a new problem to be solved
domain FEEDBACK_AVAILABLE // Triggered by SEM to CBR stating feedback available
domain WEIGHTS_UPDATED // Triggered by ADM to change influence factor weights
domain NEW_DEST // Triggered by ADM to SEM stating the new destination
domain PROBLEM_SOLVED // Triggered by CBR when it solves the given problem

SIGNAL ≡ SIGNAL ∪ PROBLEM_SOLVED ∪ WEIGHTS_UPDATED
         ∪ NEW_PROBLEM ∪ FEEDBACK_AVAILABLE ∪ NEW_DEST ∪ INFORM_ARRIVAL
```

Spec 7.3: *Signals of SEM.*

// ——————— Data Structures of SEM ———————
// Modes associated with SEM.
MODE $\equiv \{idle, pathPlanning, roadSelection, localRePlanning, running, pathCompleted\}$
mode : SEM → MODE // SEM has a mode.

destNode : SEM → NODE // Current destination node.
sourceNode : SEM → NODE // The source node.
currentEdge : SEM → NODE // Current edge stored in 'Person' Agent.
$currentEdge(a) \equiv currentEdge(parentAgent(a))$
currentNode : SEM → NODE // Current node stored in 'Person' Agent.
$currentNode(a) \equiv currentNode(parentAgent(a))$

// Stores the path suggested by the algorithm for traversal.
suggestedPath : SEM → PATH
// Stores the edge of the 'suggested path' suggested for immediate traversal.
suggestedEdge : SEM → EDGE
// Stores the final path the agent actually traverses.
takenPath : SEM → PATH

// Decides whether 'suggestedEdge' is fit for traversal, based on current road conditions.
acceptable : SEM × EDGE → BOOLEAN
// Decides whether the time required for traveling along the current edge has elapsed.
currentEdgeTraversed : SEM → BOOLEAN
// Decides whether the agent has reached the destination node.
destNodeReached : SEM → BOOLEAN
$destNodeReached \equiv currentNode(self) = destNode(self)$

Spec 7.4: *Data Structures of SEM.*

The specification of the SEM as given in Spec 7.5 formalizes the navigation algorithm at the highest natural level of abstraction and forms Level 0 of the SEM.

The SEM is initialized to be in the **idle** mode. It stays in this mode until it receives a signal NEW_DEST from the Agent Decision Module (ADM), informing that a new destination node is available to be traveled to. In this case, the abstract function INITIALIZE is called, that sets the new destination node of the SEM as passed by the signal and other supplementary information; the SEM mode is then changed to *pathPlanning*.

In the **pathPlanning** mode, the agent makes some global decisions or guesses as to which path it is going to take toward the destination. This initial path is then set as the suggested path *suggestedPath*. The idea is that generally people do not just start moving toward their

```
// ---------- Space Evolution Module (SEM) Program ----------
Program_SEM ≡
  case mode of
    idle →
      onsignal s : NEW_DEST
        INITIALIZE(currentNode, newDest(s))
        mode := pathPlanning

    pathPlanning →
      GET_PATH   // suggestedPath is updated
      mode := roadSelection

    roadSelection →
      if destNodeReached then
        mode := pathCompleted
      else
        if signalFromADM then
          HANDLE_SIGNALS_FROM_ADM
          mode := pathPlanning
        else
          GET_SUGGESTED_EDGE   // suggestedEdge is updated
          mode := localRePlanning

    localRePlanning →
      if acceptable(suggestedEdge) then
        mode := running
        currentEdge := suggestedEdge
      else
        mode := pathPlanning
      RECORD_SELECTED_EDGE

    running →
      if currentEdgeTraversed then
        UPDATE_EDGE_PERCEPTION
        FINALIZE_EDGE_TRAVERSAL
        SET_SEM_MODE

    pathCompleted →
      FINALIZE_TRIP
      mode := idle
```

Spec 7.5: *Space Evolution Module (SEM) Program.*

destination randomly, they first do some preliminary *global planning* as to which path they are going to take. This decision can be based either solely on person's past path traversal experiences stored in its memory; or by explicitly solving the problem from scratch such as by using a road map; or a combination of the two. This thus achieves a combination of *reinforcement learning* and *exploration.* The path decided on is typically a combination of a number of *influence factors* such as minimizing distance, traffic, angle; maximizing quality of experience, speed etc. In making such a decision, the person needs to know the local characteristics of the constituent edges of the potential paths; the person typically relies on information in its memory based on past experiences for the dynamic conditions of non-visible distant edges of the path (traffic, road conditions, density of people); however, for the adjacent edges that are visible, the person has access to real-time current local information. The SEM mode is then changed to *roadSelection.*

In the **roadSelection** mode, a guard is checked to see if the person has already reached its destination; if so the mode is changed to *pathCompleted.* Another check is made to see if there are any signals from the ADM, such as a change in destination etc; if so the mode is changed to *pathPlanning* indicating that the suggested path is no longer valid, and a new path needs to be re-calculated. If none of the above two conditions hold, the person then gets the next immediate edge to traverse which is set as the *suggestedEdge.* This suggested edge could be an edge of the suggested path, or a completely random choice. This incorporates *randomness* in the navigation algorithm. The SEM mode is then changed to *localReplanning.*

In the **localReplanning** mode, the person assesses the local information of the suggested edge such as the traffic flow, road conditions etc and decides whether this edge is suitable for traversal. For e.g if there is a very heavy traffic jam, the person likely wants to revise the suggested path to an alternative path. If the suggested edge is found to be acceptable, the current edge for traversal *currentEdge* is set to the suggested edge *suggestedEdge*, and the SEM mode is changed to *running*; lest the mode is changed to *pathPlanning*, indicating that a new alternative path needs to be planned. This local re-planning is important since when the decision was made on a suggested path (*suggestedPath*), the person did not have access to the current real-time local information of any of the intermediate edges, it had this information for only the edges that were adjacent to its initial position. The idea is to be able to revise previous path planning decisions on-the-fly as new information is discovered.

This incorporates real time *local re-planning* in addition to *global planning* in the algorithm.

In the **running** mode, it is emulated that the person is currently traversing the *currentEdge*. Once the person has traversed the edge, indicated by the derived function *currentEdgeTraversed*, two main actions are performed. First, using the abstract function UPDATE_EDGE_PERCEPTION, the agent specific interpretation of information for this edge, such as distance, road type, traffic, frequency, experience are updated and stored in its memory. This function ensures that the person remembers road-level decisions it has made in the past. Second, the function SET_SEM_MODE is called to decide which should be the next mode of the SEM; *pathPlanning* in case of random edge selection or *roadSelection* otherwise.

In the **pathCompleted** mode, reached from the mode *roadSelection* when the destination is reached, the abstract function FINALIZE_TRIP is called to perform some book keeping and the SEM mode is changed back to *idle*. FINALIZE_TRIP stores all the path level information that has been made available to the person by virtue of taking a path, into the memory (case-base) of the person. This ensures that the person remembers all the path-level decisions it makes, such as the path it took, along with supplementary information such as experience, road conditions etc of taking this path as a whole. This newly stored information plays an important role in subsequent path planning decisions, and in this way we achieve reinforcement learning.

7.2.2 Level 1

In this level, we apply a refinement step on Level 0 and obtain a more detailed specification of all the abstract functions of the SEM Program.

In the **idle** mode, on receiving the signal NEW_DEST from the ADM, which embeds the new destination node in the signal, the function INITIALIZE is called. INITIALIZE (Spec 7.6) updates the destination node of the agent (*destNode*) to the new node sent by the signal; sets the source node of the path to be computed to the current node of the agent; and resets the taken path to the source of the path. The mode of the SEM is then changed to *pathPlanning*.

In the **pathPlanning** mode, the person decides which path to take from the given source

```
// ———————— INITIALIZE ————————
INITIALIZE(source : NODE, dest : NODE) ≡
  destNode := dest
  sourceNode := source
  takenPath := {source}
```

Spec 7.6: *INITIALIZE.*

to the given destination. This is done by calling the function GET_PATH, which returns a path in the form of *suggestedPath*. The seemingly simple looking function GET_PATH is a complicated one, that basically incorporates the reasoning mechanism based on *reinforcement learning* implemented as a case-based reasoner and *exploration* implemented by a model-based reasoner (Spec 7.7). The function checks a predicate of the Person

```
// ———————— GET_PATH ————————
cbrDominant : SEM → BOOLEAN
cbrDominant(sem) ≡ semCBRDominant(parentAgent(sem))
GET_PATH ≡
  if cbrDominant then
    Get_Suggested_Path_Memory
  else
    Get_Suggested_Path_Explorer
```

Spec 7.7: *GET_PATH.*

agent called *cbrDominant*, (or more appropriately called *memoryDominant*) to determine whether the person likes to make decisions by recollecting facts from past experiences or by exploring new options. If the person has a memory dominant architecture, the function Get_Suggested_Path$_{Memory}$ is called, which interfaces with a case-based reasoner and bases decisions considering past experiences stored in memory; else the function Get_Suggested_Path$_{Explorer}$ is called which interfaces with a model-based reasoner and solves the problem of finding a path from scratch.

Spec 7.8 shows the details of the function Get_Suggested_Path$_{Explorer}$. It calls the function GET_SUGESTED_PATH$_{Explorer}$ which represents the model-based component of the SEM

that solves the problem of going from a given source *currentNode* to a given destination *destNode*, explicitly from scratch. This can be seen as a person opening a road map of the city and using it to compute the desired path. The suggested path *suggetedPath* is set with the path returned by the model-based component.

```
// ———————— Get_Suggested_Path_Explorer ————————
Get_Suggested_Path_Explorer ≡
  suggestedPath ← GET_SUGESTED_PATH_Explorer(self, currentNode, destNode)
```

Spec 7.8: *Get_Suggested_Path*$_{Explorer}$.

Get_Suggested_Path$_{Memory}$ (Spec 7.9) represents a hierarchical problem-solving technique, whereby, first the person relies on memory to calculate a complete path that takes him from the given source to the destination; if no such path exists, the person relies on memory to get a partial path that takes him 'close enough' to the destination and then calculates the rest of the path from scratch by using a map; if this too does not work, the person then has to solve the entire problem from scratch, exclusively using a map. This hierarchy can however be overridden in the case where the path returned form memory is one that has already been found to be unacceptable (by the function *acceptable*), in which case the person tries to explore new options.

```
Get_Suggested_Path_Memory ≡
  let pathCBR ← GET_SUGGESTED_PATH_CBR in
    if ¬empty(pathCBR) ∧ newPath(pathCBR) then
      if complete(pathCBR) then
        suggestedPath := pathCBR
      if ¬complete(pathCBR) then
        suggestedPath ← GET_SUGGESTED_PATH_Mixed(pathCBR)
    if empty(pathCBR) ∨ ¬newPath(pathCBR) then
        suggestedPath ← GET_SUGESTED_PATH_Explorer(self, currentNode, destNode)
where
  complete(p) ≡ tail(p) = destNode
  newPath(pathCBR) ≡ pathCBR ¬⊆ suggedtedPath(self)
```

Spec 7.9: *Get_Suggested_Path*$_{Memory}$.

We achieve this by first calling the abstract function GET_SUGGESTED_PATH$_{CBR}$ that interfaces with the case-based reasoner and returns a path (*pathCBR*) based solely on past experiences in the memory of the agent. There can be three cases with the returned path *pathCBR* with respect to completeness: first a complete path is returned from source to destination, second a complete path is not returned but a path 'close enough' to the destination is, and third no path is returned at all. There can be another case that the *pathCBR* is a redundant path, i.e in case an edge was found to be unacceptable in the mode *localReplanning* the CBR was called again in the mode *pathPlanning*, but it still returned a path that is (or contains) the old suggested path which is albeit unacceptable; that is the person has no knowledge of an alternative path.
If *pathCBR* is not empty, and it is a non-redundant path as determined by the derived function *newPath*, and it is also a complete path as determined with the derived function *complete*, this is set as the suggested path *suggestedPath*.
If *pathCBR* is not empty, is not redundant, but an incomplete path that takes the person closer to the destination, the agent performs *mixed reasoning*, i.e solves half the problem by using memory and the other half using a map. This is done by the abstract function GET_SUGGESTED_PATH$_{Mixed}$ which returns such a path and sets it as the *suggestedPath*.
If *pathCBR* is empty i.e, the person has neither a complete nor a partial path in memory for the given problem; or if *pathCBR* is a redundant path, i.e the person has no knowledge of an alternative path that does not include the unacceptable edge, the abstract function GET_SUGGESTED_PATH$_{Explorer}$ is invoked. GET_SUGGESTED_PATH$_{Explorer}$ returns a path based on explicit problem solving from scratch, and this path is set as the *suggestedPath*.

Thus, with the function GET_PATH, we have effectively used a hybrid reasoning system which is an integration of a Case-Based Reasoner and a Model-Based Reasoner, to emulate the decision making process of the person agent in deciding which path to take. The main idea presented here is that human reasoning is analogous to a process where we first try to solve new problems by *recollecting* past experiences; if this proves futile we try to *reuse* and *adapt* similar problem in hope of getting a solution; and only if this too proves futile, do we try to explicitly *solve* the problem from scratch using some generalized knowledge.
For example, a person wishing to travel from home to work already 'knows' the path he

is going to take since he has been taking this path for many years; he does not explicitly 'solve' the problem of which path to take, he merely *recollects from memory* the old path taken, and treads on it. On the other hand, the same person traveling from home to a new restaurant opening, first tries to see if he knows the path to this restaurant; if not, the person might recollect a point close enough to the restaurant, and then use a *map* to *compute* the rest of the path. In the worst case, when the person has no clue about how to get here, he *opens a map* to calculate the entire path.
This process of recollecting the path solely from memory is similar to using the CBR and is performed by the function GET_SUGGESTED_PATH$_{CBR}$; recollecting from memory in conjunction with using the map corresponds to using the MBR and CBR together, and is performed by GET_SUGGESTED_PATH$_{Mixed}$; and using the map for the entire path is similar to using the model-based component solely and is performed by GET_SUGGESTED_PATH $_{Explorer}$. The CBR and MBR components form the model at the next major level of abstraction and are discussed in detail in Chapters 8 and 9 respectively.

In the **roadSelection** mode, we have a suggested path (*suggestedPath*) in mind and a decision is made which edge should be followed next. First it is checked whether the person has reached the destination node, if so the mode is changed to *pathCompleted* and rest of the steps of this mode are skipped. If not, then it is checked through the predicate $signalFromADM$ whether there are any signals sent to the SEM by the ADM; if so the function HANDLE_ADM_SIGNALS (Spec 7.10) is called.

```
// ———————— HANDLE_ADM_SIGNALS ————————
HANDLE_ADM_SIGNALS ≡
  onsignal s : NEW_DEST
    INITIALIZE(currentNode, newDest(s))
  onsignal s : WEIGHTS_UPDATED
    UPDATE_WEIGHTS(s)
```

Spec 7.10: *HANDLE_ADM_SIGNALS.*

This function checks for two kinds of signals from the ADM viz: NEW_DEST and WEIGHTS_UPDATED. The NEW_DEST signal is triggered when the agent decides to abandon the current destination and picks a new destination; in this case the function INITIALIZE is called again to reset all the required information. The WEIGHTS_UPDTED

signal is triggered when the weights associated with the influence factors are changed; the abstract function UPDATE_WEIGHTS is then called to reset the new weights. Both of these signals require that the *suggestedPath* be revised since it is no longer valid; accordingly the SEM mode is thus changed back to *pathPlanning*.

If neither the destination has changed nor is there any signal from ADM, the function GET_SUGGESTED_EDGE is called that sets the next edge for traversal as *suggestedEdge*, and changes the SEM mode to the next mode *localReplanning*. GET_SUGGESTED_EDGE (Spec 7.11) basically decides whether the person follows an edge of the suggested path (Get_Suggested_Edge$_{Path}$) or follows a completely random edge (Get_Suggested_Edge$_{Random}$), by checking the predicate *goRandom*. This gives us the power to model non-deterministic and whimsical choices people sometimes make. At this level of abstraction however, we do not explicitly model how or when the predicate *goRandom* is (re)set.

```
// ---------- GET_SUGGESTED_EDGE ----------
GET_SUGGESTED_EDGE ≡
  if goRandom then
    Get_Suggested_Edge_Path
  if ≠ goRandom then
    Get_Suggested_Edge_Random
```

Spec 7.11: *GET_SUGGESTED_EDGE*.

The function Get_Suggested_Edge$_{Random}$ as specified in Spec 7.12 simply makes a non-deterministic choice in selecting an edge in the set of all edges adjacent to the person's current location (*currentNode*). This selected edge is then set as the next suggested edge *suggestedEdge*. This very simply and effectively models randomness in the navigation algorithm. The predicate *randomEdgeSelected* is set to indicate the same.

The function Get_Suggested_Edge$_{Path}$ as specified in Spec 7.13 selects the first edge of the suggested path *suggestedPath*, removes this edge from the suggested path, and sets the next suggested edge *suggetedEdge* with this selected edge.

In the **localRePlanning** mode, we have a suggested edge (*suggestedEdge*) to follow, however we reconsider the decision whether to take this edge or not. This is because the person discovers real-time local conditions of that edge (traffic flow, road condition etc) on the fly.

```
// ———————— Get_Suggested_Edge_Random ————————
randomEdgeSelected : SEM → BOOLEAN
Get_Suggested_Edge_Random ≡
    choose e in outIncidentEdges(currentNode)
      suggestedEdge := e
      randomEdgeSelected := true
```

Spec 7.12: *Get_Suggested_Edge*$_{Random}$.

```
// ———————— Get_Suggested_Edge_Path ————————
Get_Suggested_Edge_Path ≡
    let edge = firstEdge(suggestedPath) in
      suggestedEdge := edge
      remove head(suggestedPath) from suggestedPath
```

Spec 7.13: *Get_Suggested_Edge*$_{Path}$.

The GET_PATH in giving a suggested path does not consider current local information of a distant edge (but only of the edges adjacent to the initial source node) in suggesting the path since the person can have access to this information only when it is positioned on that edge, and not when it is 10 kms away from it. Thus, in this mode we examine the GEO_DYNAMIC_ATTRIBUTES of the suggested edge to decide on the acceptability of the edge, through the derived function *acceptable*. If the edge is found to be acceptable, the current edge of the agent *currentEdge* is set to suggested edge *suggestedEdge*, meaning the person has decided to traverse on this edge; and the SEM mode is set to the next mode *running*. If the edge however is not acceptable, the mode is set back to *pathPlanning* which re-calculates the path, now taking into account the local information of the current edge; for e.g. if there is a heavy traffic jam on the suggested edge, the person most likely wants to take another alternative road that is not jammed.

At the end of this mode, we do some bookkeeping activities using RECORD_SELECTED_EDGE (Spec 7.14). If the edge was found to be acceptable, the function changes the current node to undefined, indicating the person is located on an edge now and has started traversing it, and records the current time the agent has started traversing the edge; if the edge was found to be non acceptable, it records that this edge was attempted for traversal.

```
// ———————— RECORD_SELECTED_EDGE ————————
attemptedEdge : SEM → EDGE
traverseStartTime : SEM → TIME
RECORD_SELECTED_EDGE ≡
  if acceptable(suggestedEdge) then
    currentNode := undef
    // the agent starts passing the edge
    traverseStartTime := now
  else
    attemptedEdge := suggestedEdge
```

Spec 7.14: *RECORD_SELECTED_EDGE.*

In the **running** mode, the person stays within this mode until the time required to traverse the edge has elapsed; this is indicated by the derived function *currentEdgeTraversed.* Once the current edge has been traversed, we call an abstract function UPDATE_EDGE _ PERCEPTION to update the agent specific interpretation of information for this edge, such as PER_EDGE_ATTRIBUTES (distance, road type, traffic) and AC_EDGE_ATTRIBUTES (frequency, experience). The resulting updated values are typically an average of the past stored values and the current values. This function ensures that the person remembers road-level decisions it has made in the past.

The function FINALIZE_EDGE_TRAVERSAL (Spec 7.15) is called to set the current edge to undefined and current node to the tail of this traversed edge, meaning the person has now traversed the edge and its current location is that of a node; the tail node of the edge is also added to the taken path.

```
// ———————— FINALIZE_EDGE_TRAVERSAL ————————
FINALIZE_EDGE_TRAVERSAL ≡
  let node = edgeTail(currentEdge) in
    currentNode := node
    currentEdge := undef
    add node to takenPath
```

Spec 7.15: *FINALIZE_EDGE_TRAVERSAL.*

Finally the function SET_SEM_MODE is called to decide which should be the next mode of the SEM (Spec 7.16). If a random edge was *not* selected, the SEM mode is changed to *roadSelection*, since we already have a suggested path and now just need to determine the next edge to be taken. However, if a random edge was selected, the person probably is on a new path that is different from the suggested path. In this case, the mode is changed to *pathPlanning*, implying we need to recalculate the path to the destination from the current node which was reached by following a random edge.

```
// ———————— SET_SEM_MODE ————————
SET_SEM_MODE ≡
  if ¬randomEdgeSelected then
    mode := roadSelection
  else
    randomEdgeSelected := false
    mode := pathPlanning
```

Spec 7.16: *SET_SEM_MODE.*

Lastly, in the **pathCompleted** mode, all supplementary book keeping is performed with the function FINALIZE_TRIP (Spec 7.17) and the SEM mode is changed back to *idle*. There are two main tasks performed here. First is informing the ADM with the signal INFORM_ARRIVAL, and second is informing the associated CBR with the function Send_Feedback_To_CBR, that the path from the given source to the given destination has been traversed. The triggered signal to the ADM contains information such as the path taken and the current time. Informing the CBR through the function Send_Feedback_To_CBR intimates the CBR that new information is available for storing which is integrated into the existing memory (case-base) of the person; this ensures that the person remembers all the path-level decisions it makes, along with supplementary information such as experience, road conditions etc of taking this path as a whole.

7.2.3 Level 2

This level of refinement basically refines the abstract functions of GET_PATH not defined in Level 1 viz: GET_SUGGESTED_PATH$_{CBR}$ and GET_SUGGESTED_PATH$_{Mixed}$.

```
// ———————— FINALIZE_TRIP ————————
FINALIZE_TRIP ≡
  trigger s : INFORM_ARRIVAL, decisionModule(parentAgent)
    arrivalTime(s) := now
    path(s) := takenPath
  Send_Feedback_To_CBR(curProblem, pathCBR, takenPath)
```

Spec 7.17: *FINALIZE_TRIP.*

As mentioned before, GET_SUGGESTED_PATH$_{CBR}$ interfaces with the CBR of the SEM and gets a complete or a partial path from the CBR from a given source to a given destination, if such a path exists. This is analogous to the person making its decisions based solely on its memory which stores past experiences. This function defined in Spec 7.18 first checks the predicate *waitingForSignal* to determine whether the SEM is already waiting to hear back from CBR or it can proceed with sending a new problem. If the predicate is false, the problem is sent to the CBR using the function SEND_NEW_PROBLEM_TO_CBR and the predicate is set to true, indicating that a solution is awaited. The SEM then loops over this function until a solution is returned by the CBR indicated by the signal PROBLEM_SOLVED, in which case the solution embedded in this signal is returned to the SEM (which forms the *pathCBR*), and the predicate is set to false.

```
// ———————— GET_SUGGESTED_PATH_CBR ————————
GET_SUGGESTED_PATH_CBR ≡
  if ¬waitingForSignal then
    SEND_NEW_PROBLEM_TO_CBR(currentNode, destNode, closeness)
    waitingForSignal := true
  if waitingForSignal then
    onsignal s : PROBLEM_SOLVED
      return solution(s)
      waitingForSignal := false
where
  closeness = closeness(self)
```

Spec 7.18: *GET_SUGGESTED_PATH$_{CBR}$.*

The parameter *closeness* of SEND_NEW_PROBLEM_TO_CBR is a derived function that

is used to indicate how 'incomplete' a partial path can be. If only an exact match from source to the destination is expected, closeness is set to 0. For partial paths, closeness is non-zero and indicates the maximum number of hops by which the destination of the partial path can be away from the final destination. Naturally, when a non-zero value of closeness is passed, the CBR first tries to retrieve a path that is complete, and then monotonically increases the number of hops by which the final destination can be away from the partial path destination.

GET_SUGGESTED_PATH$_{Mixed}$ as given in Spec 7.19 represents a form of hybrid reasoning. It takes the partial path returned by GET_SUGGESTED_PATH$_{CBR}$, which is passed to it as a parameter *partialPathCBR*. It then calls the model-based component of the SEM GET_SUGGESTED_PATH$_{Explorer}$ to compute the rest of the path, i.e, a path from the tail of the *partialPathCBR* to the final destination; this path is the *partialPathExplorer*. The concatenation of the above two paths gives the complete path, and forms the desired mixed path (*pathMixed*). Next, we invoke the function GET_SUGGESTED_PATH$_{Explorer}$ to return a path based on explicit problem solving from scratch, as *pathExplorer*. A comparison between the complete path returned by the explorer and the path formed by concatenation is performed by the abstract function *superior*(*pathMixed*, *pathExplorer*) that checks to see whether *pathMixed* is within an acceptable interval of inferiority as compared to *pathExplorer*; if so *pathMixed* is returned as the suggested path, else *pathExplorer* is returned as the *suggestedPath*. This comparison is important since by concatenation of partial paths one might end up with a path that does reach the destination but is a very undesirable path.

GET_SUGGESTED_PATH$_{Mixed}$(*partialPathCBR* : PATH) ≡
  let *partialPathExplorer* ← GET_SUGGESTED_PATH$_{Explorer}$(*self*, *tail*(*partialPathCBR*), *destNode*)
  in
    let
      pathMixed ← *concat*(*partialPathCBR*, *partialPathExplorer*)
      pathExplorer ← PATH_EXPLORER(*self*, *currentNode*, *destNode*)
    in
      return *superior*(*pathMixed*, *pathExplorer*)

Spec 7.19: *GET_SUGGESTED_PATH*$_{Mixed}$.

Although in the mixed approach we currently restrict to finding a partial path from the source to the destination, one should be cognizant that the converse case, i.e, finding a partial path from the destination to the source, can also be easily incorporated into the framework.

The abstract functions SEND_NEW_PROBLEM_TO_CBR and SEND_FEEDBACK_TO_CBR are the ones that actually send the problem and the feedback to the CBR, by first initializing them and then triggering the CBR with them. These form the next level of refinement of the SEM model — Level 3. The details of these functions however are not discussed here, since the working is pretty basic and self-explanatory, but can be found in Appendix A.

7.2.4 Our Navigation Approach in Comparison to Related Work

The navigation problem is a crucial one that finds applications in various fields, most importantly Intelligent Transportation Systems, and GPS systems. While there is abundant literature highlighting the approaches taken in these fields, our approach is significantly different and offers numerous advantages in a number of ways.

Our path planning algorithm effectively combines Case-Based Reasoning, Model-Based Reasoning and Human Perception in a novel manner. We take a holistic view of the cognitive decisions a person makes while moving in an urban landscape; the path might not be a globally optimal one, but it is a good-enough one that mimics human reasoning and decision-making more closely. Specifically, the fortes of our algorithm include: (1) it takes reinforcement learning into account (positive and negative) (2) it combines global planning with real-time local re-planning (3) it takes subjective human perception into account (4) the shortest path algorithm takes a number of factors into account (not just distance or travel time) whose influence in the overall preference can be dynamically changed (5) it effectively makes a distinction between people recollecting paths from the memory vs. re-computing paths using maps (case-based reasoning vs. model-based reasoning) (6) it can make both road level and path level decisions, by taking cases of *roads* (in MBR) and cases of *paths* (in CBR) into account respec. (7) it can model random and non-deterministic decisions that people may sometimes make.

We have presented the contention that people do not move in an urban environment based on how it physically is, but how they 'perceive' it to be. This perception is incorporated by dividing the environment into Geographic Environment and Subjective Environment, as explained in Chapter 6; both these categories then play an integrated role in shaping decisions. To the best of our knowledge, there has been no work presented so far that takes cognition of geography into account while path planning.

Our algorithm relies on the CBR to immediately return solutions for a shortest path problem from a given source to destination, if any. This is similar to a person trying to recall solutions from memory rather than re-computing them.

If this proves futile, we rely on the CBR to return a path that is close-enough to the destination, and then use the MBR to compute the remaining path. This is similar to people relying on memory to recall a partial path, and then computing the remaining from a map. If this proves fruitless, the problem is then relegated to the MBR, which uses a shortest path

algorithm, based not only on distance, but a number of other factors (road type, traffic, road conditions, familiarity, experience, angle, # of intervening roads etc) that determine the preference or cost of a path. To the best of our knowledge, no other approach has taken into account such a vast number of factors in path planning. Most algorithms typically seek to minimize only the travel time, or the distance of travel.

Bing Liu et al [69] propose an algorithm based on case-based reasoning, knowledge-based approach and Dijkstra algorithm for route finding. However, they restrict only to distance or travel time for cost calculation. Their interaction between the CBR and the KBR is not well-defined and produces inefficient results, when the reasoner returns partial paths. Our algorithm uses a number of cost factors, the influence of which can be changed dynamically. Also the interaction between our CBR and MBR is well-defined that produces efficient results.

Anwar and Yoshida [2] integrate cases, knowledge and object-oriented road network for route finding. However, their approach is restricted to only a very specific kind of road network composed of *must be passed* links, and not applicable to general networks. Their approach does not update the cases dynamically and hence the reasoner cannot learn. Our approach integrates new knowledge into the CBR case-base, and the MBR cases every time a road or a path is taken; this ensures both the reasoners learn and become competent over time.

The authors of [68] present another approach that combines cases, knowledge and a Dijkstra like algorithm. However, they restrict learning and experience to road levels only and do not consider path level learning and experiences. They ignore the merits of a case-based reasoner in returning a path when an exact match can be found. Further their algorithm first prunes off the search space based on major and minor roads and then applies the Dijkstra's algorithm; this forces a person to takes major roads that might be longer when there can be minor road short-cuts. Our approach on the other hand considers both road and path level preferences, by using road cases in the the Dijkstra algorithm and using path cases in the case-based reasoner, respectively. We do not partition the graph based on major and minor roads; instead we dynamically change the weight associated with factor 'road type' to increase or decrease the preference of major/minor roads.

7.3 Target Selection Module (TSM)

The Target Selection Module (TSM), implemented as an ASM Agent, is responsible for monitoring potential targets on the routes taken by the agent, and for selecting attractive targets based on selection criteria. This leads to the creation of the *crime occurrence space* of the agent. We essentially base the logic represented by the TSM on the *Basic Model of Target Selection* as given by the Brantinghams [10], [9]. The model views the target selection process as an information processing model. It states given that there is a motivated offender, the search process is a sequence of spatial decisions in which the objective environment is perceived and evaluated. The criminal carries a template of target selection in its mind which is learnt by experience or by social interactions. In contemplating a new target selection decision, it compares the characteristics of this potential target (spatial, temporal, legal etc.) against its own template, decides on the suitability of the target and either rejects or accepts it for victimization. The template is established with experience and time and become relatively fixed and self-enforcing. Due to the multiplicity of targets and victims, many potential crime selection templates could be constructed for a given criminal.

For the process of target selection, we use a hybrid reasoning mechanism composed of case-based reasoning (CBR) and model-based reasoning (MBR); the two components are represented as ASM agent and submachine respectively. At this level of abstraction however, none of these two components are refined. The case-based reasoner has a *case-base*, which is a kind of memory, that stores previous target templating and target selection decisions an agents has made; in selecting a new target, an agent then merely matches the characteristics of the potential target against victimized targets in its memory and decides whether it is a 'good' or a 'bad' target. The MBR component on the other hand uses an algorithmic approach to explicitly determine the suitability of the targets.

7.3.1 Level 0

We first introduce some domains and data structures with the TSM. The domain CRIME_TYPE identifies different types of crime that can be modeled, such as car theft, burglary, shop lift etc.; it can be extended to include other crimes. The domain LOCATION can

either be a NODE or an EDGE. The domain TARGET represents a criminal opportunity. The target could be PASSIVE_OBJECT such as a stationery car, or it could be an ACTIVE_OBJECT such as a person. There are additional functions associated with a TARGET such as its location, and the type of criminal opportunity it represents. We then associate derived functions *potentialTargets* with NODES and EDGES, that gives the targets located on that node or edge.

```
// ———————— Domains for the TSM ————————
domain CRIME_TYPE
carTheft :→ CRIME_TYPE
shopLift :→ CRIME_TYPE
robbery :→ CRIME_TYPE

// Target
domain LOCATION ≡ NODE ∪ EDGE
domain PASSIVE_OBJECT
domain ACTIVE_OBJECT
domain TARGET ≡ PASSIVE_OBJECT ∪ ACTIVE_OBJECT

crimeType : TARGET → CRIME_TYPE
location : TARGET → LOCATION

// Functions added to Edge and Node
potentialTargets : EDGE → TARGET − set
potentialTargets : NODE → TARGET − set
```

Spec 7.20: *Domains of TSM.*

Next, to refine the abstract rule TSM, we enumerate the functions that it uses. This is formulated in Spec 7.21 in a self-explanatory manner.

Finally, the TSM Program gives the working of the TSM, and is shown in spec 7.22. The TSM is initialized to be in the *observing* mode; this mode does not perform anything unless a signal CRIMINAL_MOTIVATED is received from the ADM. This signal indicates that the person is criminally motivated and will indulge in criminal activities. Upon receiving this signal, all targets that are adjacent or visible to the criminal from its current location are recorded as $potentialTargets$ by the abstract function GET_POTENTIAL_TARGETS. The mode is now changed to $targetTemplating$.

```
// ---------- Functions of TSM ----------
MODE ≡ {observing, targetTemplating, targetSelection}
mode : TSM → MODE  // TSM has a mode

// Kept in Profile
criminality : TSM × CRIME_TYPE → PROBABILITY// Specialty of criminal for diff crimes

// Kept in Working Memory
potentialTargets : TSM → TARGET − Set  // All observed targets.
goodTargets : TSM → TARGET − Set  // Targets 'templated' as 'good'
selectedTargets : TSM → TARGET − Set  // Targets eventually victimized

// Auxiliary
currentLocation : TSM → LOCATION  // Current node or edge of the person
```

Spec 7.21: *Functions of TSM.*

```
// ---------- TSM Program ----------
TSMProgram ≡
  case mode of
    observing →
      onsignal s : CRIMINAL_MOTIVATED
        if currentLocation ≠ undef then
          // Get all the targets located 'around, on' current Location.
          GET_POTENTIAL_TARGETS(currentLocation)// Sets potentialTargets.
          mode := targetTemplating

    targetTemplating →
      // Compare environmental cues of target against your own template
      // Based on templating, categorise targets as 'good' or 'bad'
      // This process can be based on memory/cases(CBR) or explicitly executed(MBR)
      TARGET_TEMPLATING(potentialTargets)// Sets goodTargets.
      mode := targetSelection

    targetSelection →
      // Victimize good targets, reinforce target template.
      SELECT_TARGETS(goodTargets)// Sets selectedTargets
      mode := observing
```

Spec 7.22: *TSM Program.*

In the *targetTemplating* mode, the person assesses the environmental cues about the potential target such as its physical, spatial, temporal, cultural, legal, psychological characteristics against its own *target template*, and based on this comparison categorizes the potential targets into different categories such as 'good' and 'bad'. This is done by the abstract rule TARGET_TEMPLATING. The cues that are used to assess the suitability of the targets form the *target template*. As experiential knowledge grows, the person learns which individual cues are associated with 'good' targets; for e.g a car parked at a position which is within the criminal's awareness space forms a good spatial cue. These cues, cue clusters, and cue sequences (spatial, temporal, social, and so on) are the *template* against which targets are compared and accepted or rejected for victimization. This mode thus compares the properties of the potential target against the person's *target template* and categorizes the targets as good or bad; the function $goodTargets$ is set and the mode is changed to $targetSelection$.

This entire process of target templating as performed by TARGET_TEMPLATING can be based on past learnt experience (CBR) for repeat offenders, or performed explicitly from scratch (MBR) for first time offenders, or a combination of the two. The CBR typically stores all the previous target templating decisions the person has made, i.e for the given target which template it used and what was the assessment (good, bad). For each target template and its assessment, the CBR could even store the consequence of target selection (successful, failed).

In the $targetSelection$ mode, it is emulated that the $goodTargets$ as determined in the previous step are potentially victimized. This is done by the abstract rule TARGET_SELECTION. This rule could even record the outcome of target selection, and accordingly reinforce the *target template*. Thus once the template is established, it becomes relatively fixed and influences future searching behavior, thereby becoming self-reinforcing. Finally, the mode is changed back to observing.

It should be noted that the entire SEM program is performed for a given location of the agent, i.e the location of the agent should not change until target selection has been performed. This thus requires synchronizing the working of the TSM with the SEM. We assume there is an implicit clock that orchestrates this synchronization such that while the TSM performs its steps, the SEM does not change the current location of the person.

In the next level of refinement, i.e Level 1, the abstract rules GET_POTENTIAL_TARGETS, TARGET_TEMPLATING and SELECT_TARGETS are stated. We have not yet broached the details of this level. For the purpose of this thesis, we select and victimize targets non-deterministically provided they are within the criminal's awareness space. It is anticipated future work will involve detailing these abstract functions. This would entail understanding the intricacies of target templating and selection more thoroughly, formulating the various factors and variables that come into play and how they correlate with each other; this step needs to be carried out in consultation with the criminologists.

7.4 Agent Decision Module (ADM)

The Agent Decision Module (ADM) monitors the working of the TSM and SEM and provides relevant inputs to the two modules. It decides on 'what to do' and then relegates the decision to the TSM or the ADM on 'how to do it'. For e.g., it decides that the agent should go from home to work, and then gives this 'goal' to the SEM. The decisions are based on agent's motivations, the current state of the agent, and the information in the memory. Motivations are long-term goals (earn livelihood, greed), that in turn give rise to short term goals (go from home-to-work, car theft), which are then passed to one of the two modules for realization.

7.4.1 Level 0

We use abstract functions to interface with motivations; currently, we associate two broad categories of motivations with abstract functions viz: the motivation *routine activity* with the abstract function ROUTINE_ACTIVITY and the motivation *criminal propensity* with the abstract function CRIMINAL_PROPENSITY. Routine Activity basically decides the level of motivation a person has to carry on daily activity routines; this thus indirectly controls the working of the SEM. Criminal Propensity gives the current level of motivation of the person for criminally disposed activities which decides if the person will indulge in committing crimes; this thus indirectly controls the working of the TSM. Both of these motivations can be based on a number of other finer grains of motivations and demographic factors of the person, at this stage however we abstract away from such complicated functionality.

Currently in our model, at the given level of abstraction, motivations are typically always persistent. This implies that for non-criminals the motivation *routine activity* is always above a threshold level and the motivation *criminal propensity* is always below the threshold level, such that person never commits a crime; for criminals, both the motivations *routine activity* and *criminal propensity* are true, such that the criminal is always criminally disposed.

We can then associate two kinds of rules with the ADM Program, one that monitors the working of the SEM based on the motivation of *routine activity*, and the other that controls

the working of the TSM based on the motivation of *criminal propensity*. The ADM at the first level of abstraction is shown in Spec 7.23.

```
// ---------- ADM Functions ----------
// Motivation
routineActivity :→ MOTIVATION
criminalPropensity :→ MOTIVATION

// Kept in Profile
motivations : ADM → MOTIVATION − Set  // Person has motivations

ADMProgram ≡
  if ROUTINE_ACTIVITY(routineActivity, self) ≥ threshold(routineActivity)
   then
     ADM_SEM_MONITOR
  if CRIMINAL_PROPENSITY(criminalyPropensity, self) ≥ threshold(criminalyPropensity)
   then
     ADM_TSM_MONITOR
```

Spec 7.23: *ADM Program.*

7.4.2 Level 1

We first refine the abstract rule ADM_SEM_MONITOR. Once the person has the routine activity motivation above a certain threshold, this function is called. Its main responsibility is to provide the SEM the next destination node to travel to; which forms the 'goal' that is communicated to the SEM. This thus requires that each person should be associated with a schedule. We refine the function ADM_SEM_MONITOR by first associating a schedule with it (Spec 7.24).

Each person has two kinds of schedules, *specialSchedule* and *regularSchedule*. The regular schedule is an everyday schedule that is continually repeated and is differentiated by week day and weekday; what we do on weekends is different from what we do on weekdays. The special schedule can be used to override the regular schedule, and is specified for a date range; e.g. people going on vacations. For any given day, a person has a daily schedule as given by *dailySchedule*; this is nothing but a set of probabilistic destination nodes ($toNodeSet$) associated with the time of the day ($timeOfDay$). The $toNodeSet$ assigns a probability for

```
// ———————— PERSONAL SCHEDULE ————————
// -PROBABLE_DEST-
toNode : PROBABLE_DEST → NODE
prob : PROBABLE_DEST → PROBABILITY
// -APPOINTMENT-
timeOfDay : APPOINTMENT → TIMEOFDAY
toNodeSet : APPOINTMENT → PROBABLE_DEST – Set
// -SCHEDULE-
fromDate : SCHEDULE → DATE
toDate : SCHEDULE → DATE
dayType : SCHEDULE → DAYTYPE  // weekday, weekend
dailySchedule : SCHEDULE → APPOINTMENT – Set
// -PERSONAL_SCHEDULE-
regularSchedule : PERSONAL_SCHEDULE → SCHEDULE – Set  // fromDate=toDate=undef
specialSchedule : PERSONAL_SCHEDULE → SCHEDULE – Set  // fromDate not undef
```

Spec 7.24: *Personal Schedule used by ADM_SEM_MONITOR.*

each destination node; for e.g. at 2 pm a person may either go for lunch with a 80% chance, or go back home with a 20% chance.

The rule ADM_SEM_MONITOR is responsible for continually monitoring time, and checking the schedule of the person to see if there is a new destination available corresponding to the new time change. It may then decides whether to inform the SEM with this new destination. In order to do so, we associate some basic functions with this rule as given in Spec 7.25; these are self-explanatory.

The working of the rule ADM_SEM_MONITOR at the first level of abstraction, level 0, is shown in Spec 7.26. It is specified as a submachine that is a sequence of four steps; although for the sake of lucidity the steps are associated with controlled modes, they are all still executed in sequence one after the other. The ASM_SEM_MONITOR submachine is initialized to be in *monitor* mode.

In the *idle* mode, the time as derived from the abstract domain TIME, is continually monitored using the abstract function MONITOR_TIME to see if there is a change in current time from the last recorded time, and if so the mode is changed to *calculate*. The granularity of change in time is person dependant and could be a change by the hour, by the minute etc. This thus decides whether the person makes decision by the minute or hour.

// ——————— Functions on ADM used by ADM_SEM_MONITOR ———————
domain MODE ≡ {*monitor*, *calculate*, *decide*, *inform*}
mode : ADM → MODE // mode of ADM
// —Kept in Profile—
schedule : ADM → PERSONAL_SCHEDULE
// —Auxiliary—
nextDest : ADM → NODE // next destination calculated
arrived : ADM → BOOLEAN // whether the person has reached its current destination
timeChanged : ADM → BOOLEAN // time has changed
lastTime : ADM → TIMEOFDAY // records last time
inform : ADM → BOOLEAN // inform the SEM?

Spec 7.25: *ADM Functions used by ADM_SEM_MONITOR.*

Also monitored in this mode is a signal from the SEM which indicates whether the person has reached its current destination, this is recorded in the predicate *arrived.*
In *calculate* mode, the function CALCULATE_NEXT_DEST uses the *schedule* of the person to decide the next destination based on probabilistic choices. The function *nextDest* is set with this chosen destination, and the mode is changed to decide.
In *decide* mode, it is checked whether a destination was available from the schedule. If so, the function DECIDE_TO_INFORM determines whether the agent should be informed with this new destination by setting the predicate *inform*; the mode is now changed to inform. If no destination was set, the mode is changed back to monitor. Typically in making the decision on whether to inform the agent with the new destination, if the person has already reached its destination, *inform* is set to true; else a non-deterministic choice is made.
In the *inform* mode, the predicate *inform* is checked to determine if the person should indeed be informed of the new destination, if so the signal NEW_DEST is triggered to the SEM, and the mode is changed back to monitor.

Next, we refine the abstract rule ADM_TSM_MONITOR. Once the person has the *criminal propensity* motivation above a certain threshold, the rule ADM_TSM_MONITOR is called. Its main responsibility is to provide certain 'goals' to the TSM. At this level of abstraction, ADM_TSM_MONITOR provides minimal functionality; it merely triggers the TSM with the signal CRIMINAL_MOTIVATED indicating the fact that the TSM should proceed with the process of target selection. Upon receiving this information as a 'goal', the TSM then

```
// ———————— ADM_SEM_MONITOR ————————
ADM_SEM_MONITOR ≡
  case mode of
    monitor →
      // Record the fact agent has arrived at a dest.
      onsignal s : INFORM_ARRIVAL
          arrived(self) := true
      MONITOR_TIME(self)
      // Keep monitoring time to see if it changes(e.g. morning to noon, 3pm to 4pm).
      // If changes, set mode to calculate.
    seq
    calculate →
      // Probabilistically choose the next destination in the schedule.
      // Set nextDest.
      CALCULATE_NEXT_DEST(self)
      mode := decide
    seq
    decide →
      if nextDest(self) then
          // Decide whether to inform the agent or not - Set Inform predicate
          // Usually if arrived = true inform, else decide.
          DECIDE_TO_INFORM(self)
          mode := inform
      else
          // No schedule available. Go back to Monitoring time.
          mode := monitor
    seq
    inform →
      if inform(self) then
        // triggers the SEM with the New Destination.
        trigger s : NEW_DEST, spaceModule(parentAgent)
          newDest(s) := nextDest(self)
      mode := monitor
```

Spec 7.26: *ADM_SEM_MONITOR.*

proceeds with executing its functionality. This can be seen in Spec 7.27.

```
// ———————— ADM_TSM_MONITOR ————————
ADM_TSM_MONITOR ≡
  // triggers the TSM to initialize its execution.
  trigger s : CRIMINAL_MOTIVATED, targetModule(parentAgent)
```

Spec 7.27: *ADM_TSM_MONITOR.*

In the next level of refinement, Level 2, the abstract functions MONITOR_TIME, CALCULATE_NEXT_DEST, DECIDE_TO_INFORM are detailed. The abstract domain TIME and the relevant functions defined on it are also explicitly specified. We do not show these here, but refer the reader to AppendixA.

PART III

REFINED MODEL

Chapter 8

Reasoning and Learning

We use a mechanism of hybrid reasoning — based on case-based reasoning and model-based reasoning — to emulate the problem solving and decision making process of the agents. This chapter is devoted to analyzing the case-based reasoning component of the hybrid framework. We begin with Section 8.1 which is intended to provide the reader an introductory knowledge of Case-Based Reasoning and we conclude with Section 8.2 which explains how we develop, formalize, and integrate a case-based reasoner into our framework, to solve the problem of path planning.
To the best knowledge of the author, there has been no prior attempt in establishing a formal specification and executable semantics of the case-based reasoning process. Thus, this sub-part of our work, by itself, forms an original contribution.

8.1 Case-Based Reasoning (CBR)

"Case-Based Reasoning is an approach to reasoning whereby instead of solving the problem from scratch using rules, it is solved by remembering previous similar situations called cases, and by reusing this knowledge from past experiences" [1]. Thus, in CBR reasoning is based on remembering.
CBR is based on two tenets about the world — the first tenet is that the world is regular: similar problems have similar situations; the second tenet is that the types of problems an agent encounters tend to recur [67]. Thus, a CBR can be used to solve routine problems, or

novel problems by adapting the solution of similar problems.

The foundational motivations of CBR stem primarily from two fields, (1) from Cognitive Science - to model human reasoning and learning; (2) from AI - to develop pragmatic technology [67]. The approach of Case-Based Reasoning relates to other areas such as memory-based reasoning, analogical reasoning, exemplar-based reasoning [67],[1]. Case-Based Reasoning can also be seen as a formal computational model of problem solving based on memory organization and reminding [72].

Learning plays a central role in Case Based Reasoning. Learning is an emergent behavior that arises from the case-based reasoner's normal functioning [65].
A CBR learns from past experiences, and these experiences then supplement further reasoning; forming an intertwined cycle. Once a CBR solves a problem, the solution is retained as a *case*, and this case becomes available as a potential solution to similar problems in the future. Thus, complementary to the principle of *reasoning by remembering* is the principle that *reasoning is remembered* [67]. CBR is an approach to incremental, sustained learning [1]. Learning in CBR systems is by both *success-driven and failure-driven* [67].
Case-based reasoners also become more *competent* and *efficient* over time [66]. They become more efficient by remembering old solutions and adapting them, rather than deriving solutions from scratch using rules. They become more competent by deriving better answers with experience, than they would without experience.

Case-Based Reasoners can be of two types — *problem solving reasoners* or *interpretive reasoners* [66]. Interpretive Reasoners use prior cases for classifying or characterizing new problems. Problem Solvers use case-based reasoning to suggest solutions to the given problem based on past experiences. Problem-Solving CBR can be further divided into *derivational* and *transformational* CBR. In transformational CBR, past solutions are used directly; in derivational CBR, the problem-solving process by which solutions are derived are used.

For practical applications of CBR, it may be implemented as *Autonomous Systems* — that solve problems solely by themselves, *Human-Machine Systems* — that work along with people to solve problems, *Embedded/Hybrid Systems* — that in addition to CBR, use multiple knowledge sources, reasoning methods [66].

Case-Based Reasoning has a wide array of applications in fields of medicine, law, automotive, robotics etcetera [65], [67], [82].

8.1.1 Why Use CBR?

Fortes of CBR as compared to other reasoning mechanisms are many-fold: case-based systems allow for intra-domain reasoning; they save time by re-using old solutions, rather than deriving solutions from scratch; case-based reasoning provides *creative thinking* by solving problems by using similarity of past experiences; they emulate the *natural way* that human beings solve problems; case-based reasoners integrate learning and reasoning hand-in-hand; case-based reasoners facilitate both *positive and negative reinforcement learning* [65], [67].

Pertinent to our context, the use of case-based reasoning is justified and supported by three tenets as highlighted below:

The primary motivation for using a CBR based approach is that it captures the very essence of human reasoning, in the most intuitive and natural way. The cognitive process by which agents perform reasoning and learning is based very closely to the philosophical foundations of case-based reasoning. We learn from past experiences and use these very experiences in solving new problems rather than re-solving the problem from scratch. Virtually whenever there is a case available to reason from, people will *match* it, *retrieve* it, *adapt* it, *evaluate* it, *use* it, and finally *store* it as a solution — the process is analogous to that of a CBR. The same thought is re-instated in [82] - *"People really don't think all that much, they remember. First, we remember the the things we do, including the thinking we do. Second, most of the time we don't need to think , we just have to remember what we thought before."* The authors explain that *human beings are not systems of rules, but libraries of experiences*, and people *re-use* these past experiences for problem solving, rather than *re-inventing* the solution by original thinking — *"Real thinking has nothing to do with logic at all. Real thinking means retrieval of the right information at the right time."*

Secondly, with case-based reasoning, we effectively incorporate *Learning* in our model. In CBR, both reasoning and learning go hand in hand. Complementary with the principle of *reasoning by remembering* is the principle that *reasoning is remembered* [67]. We mimic *Behavioral Reinforcement Learning* by using the candidacy of past cases as potential solutions to current problems. If an agent has taken a certain route X from A to B before, this route gets recorded as a case. Next time, the agent wishes to travel from A to B again, this stored route X gets retrieved as a potential solution. Hence, once a person does something, the more he/she does it again — old habits die hard!

By storing the *outcome* of the proposed solution in the case, both *positive and negative reinforcement learning* can be incorporated. If the outcome of the proposed solution/case had a negative reinforcement — the person took a certain path X from A to B, but got robbed along this path — this gets recorded as a negative outcome for the case X. Next time the person wishes to move from A to B, path X will be retrieved as a negative case, and will prevent the person from taking this path. This is similar to 'learning the moral of the story' [82].

Thirdly, using CBR helps in modeling *idiosyncrasies* of people. Some people exhibit certain 'exceptional' behavior, which clearly cannot be captured by using general rules, since rules portray 'generalized behavior' of people. These exceptions can then be stored as 'cases' in the case-base, after which behavior can be emulated.

8.1.2 Integrations of Case-Based Reasoning

It is generally believed that complex problems can be easier solved with *Hybrid Systems.* The goal of hybrid systems is to tap information from different knowledge representations, and augment the positive aspects of the integrated formalisms while simultaneously minimizing their negative aspects. A hybrid model of CBR provides a flexible and comprehensive model, by integrating multiple levels of knowledge — specific situations from past experience and generalized domain knowledge [72].

CBR can be integrated with other reasoning modalities and computing techniques, including rule-based reasoning (RBR), constraint-satisfaction problem (CSP), model-based reasoning (MBR), genetic algorithms and information retrieval [73]. Integrated approaches increase accuracy, efficiency, problem-solving strength, combine the advantages of different approaches, and help attain a more complete and cognitive problem-solving model.
The hybrid architecture can be trichotomized as follows: (1) Master-Slave: other reasoning methods support the CBR component; (2) Slave-Master: CBR component supports other reasoning methods; (3) Collaborating: CBR and other reasoning components are balanced in their roles [73].

One such hybrid system is an integration of a Rule-Based and a Case-Based System.
In [55], authors use a case-based system to improve the accuracy and efficiency of a Rule-Based system, that the system could not have achieved with its rules alone. First, the RBR

is used to find an approximation to the problem, but if the problem is similar to an exception stored as a case, then the aspect is modeled after the case rather than the rules. To facilitate a decision process, an agent may use a *rule dominant architecture* — where the rule-based component plays the primary role and is supplemented by the case-based component; or a *case dominant architecture* — where the case-based component prevails and the rule-based component plays a supportive role; or a *balanced architecture* — where the roles of both the components are equal [79]. Another categorization of such a system can be on the basis of integration being *efficiency-improving* or *accuracy-improving* [55]. In efficiency-improving integrations, cases and rules are derived from each other, and the efficiency of the integrated system exceeds that of using cases or rules alone. Those that are dependent can be further classified according to which knowledge source was derived from which. Usually most such integrations have cases derived from rules; cases are records of how rules were applied to particular situation encountered previously. In accuracy-improving systems, cases and rules are independent, which once integrated yield higher accuracy. The primary motivation behind such systems is to increase accuracy, by tapping into as many knowledge sources as possible, and incorporating their strengths.

Successful examples can be found on the integration of CBR with other techniques in a wide array of fields [79], [55], [69], [73].
Our approach uses a hybrid framework composed of a model-based and a case-based component. The principles discussed above for a general CBR hybrid and a specific RBR-CBR hybrid can both be applied to our approach. The proposed approach is explored further in the following Section.

8.2 Our Approach: Integrating CBR into the Framework

This section explains how we incorporate case-based reasoning into our existing framework. Section 8.2.1 discusses the need for integration of case-based reasoner with a model-based reasoner. In order to do so, we first provide a high-level ASM specification of a generalized abstract reasoner in Section 8.2.2. This abstract reasoner can then be instantiated to a concrete reasoner for a given problem, as per the need of the application; in Section 8.2.3, we instantiate the abstract reasoner for solving the problem of path planning.

8.2.1 A CBR-MBR Hybrid System

We use a hybrid system which is an integration of a Case-Based Reasoner (CBR) and a Model-Based Reasoner (MBR), to emulate the decision making process of the person agent. Typically, each kind of problem — path finding, selecting targets — will have its own CBR and MBR components. For the problem of path planning, the model-based component is called the *path explorer* and is discussed in Chapter 9, and the case-based component is the case-based reasoner being discussed in this chapter.

The role of CBR is analogous to humans solving problems by remembering and recollecting past experiences stored in their memory. The role of MBR is analogous to humans explicitly solving problems right from scratch, this thus typically represents an algorithmic computation.

The relation between the two components is *hierarchical* in nature. Typically, the case-based reasoner is called first to provide a solution; if a satisfactory solution is not returned, the case-based component in conjunction with the model-based component is invoked; and as the last resort the model-based component solely is invoked. However, the hierarchy is not fixed for all person agents, and depends on their personal preferences (stored in profile). The main idea is that human reasoning is analogous to a process where we first try to solve new problems by recollecting past experiences; if this proves futile we try to reuse and adapt similar problem in hope of getting a solution; and only if this too proves futile, do we try to explicitly solve the problem from scratch using some generalized knowledge. Thus, model-based component and case-based component have complementary strengths — model-based systems use *generalized knowledge* to assist in solving a new problem; case-based systems on the other hand, store past experience derived from this knowledge as *specific episodes* for individual problem solving. The cases can then be seen as specific applications of general knowledge.

For example, a person wishing to travel from home to work already 'knows' the path he is going to take since he has been taking this path for many years; he does not explicitly 'solve' the problem of which path to take, he merely *recollects from memory* the old path taken, and treads on it. On the other hand, the same person traveling from home to a new restaurant opening, first tries to see if he knows the path to this restaurant, or a similar old path that when modified a bit will take him to this restaurant; if not, the person might

recollect a point close enough to the restaurant, and then use a *map* to *compute* the rest of the path. In the worst case, when the person has no clue about how to get here, he *opens a map* to calculate the entire path.
This process of recollecting the path from memory is similar to using the CBR of the hybrid system, recollecting from memory in conjunction with using the map corresponds to using the MBR and CBR together, and using the map for the entire path is similar to using the model-based component solely.

8.2.2 High-Level Specification of an Abstract CBR

In this section we devise a high-level ASM specification of an abstract case-based reasoner. The reasoner is abstract in the sense that it is a generalized reasoner independent of any application or domain-specific requirements. The reasoner uses powerful abstractions for processes that otherwise require sophisticated algorithms and are application-specific. Step-by-step refinements of this generic reasoner can be used to build a concrete, specialized reasoner for a given problem and domain.
To the best knowledge of the author, there has been no prior attempt in establishing a formal semantic specification of a case-based reasoner. Thus, this sub-part of our work in itself forms an original research contribution.
We describe the case-based reasoning process in a step-by-step manner, adapted from popular literature [66] [1] [65], and alongside show the respective formal ASM specification.

Top-Level Architecture

First and foremost, a case-based reasoner has a *case base*, which is the repository or the library of all past experiences/cases. Given that there is a new *problem* to solve, the basic cycle of a case based reasoner is a four-step sequential process. The **Retrieve** phase retrieves a case from the case base that best matches the problem; this case gives the *ballpark solution.* The **Reuse** phase simply reuses or adapts the retrieved solution (ballpark solution) to better suit the problem description; this is the *final solution* that is returned to the user. The **Evaluate** phase takes *external feedback* from the user (system) of applying this proposed solution to the problem, assesses the outcome of selecting this case as a potential solution, determines if the solution was faulty, and repairs it if required; this forms the *repaired*

solution. Lastly, the **Retain** phase extracts new information embedded in the external feedback, and integrates this *extracted information* into the case base. Each of the above four top-level phases can be broken down into a number of sub-tasks. Figure 8.1 illustrates the CBR process.

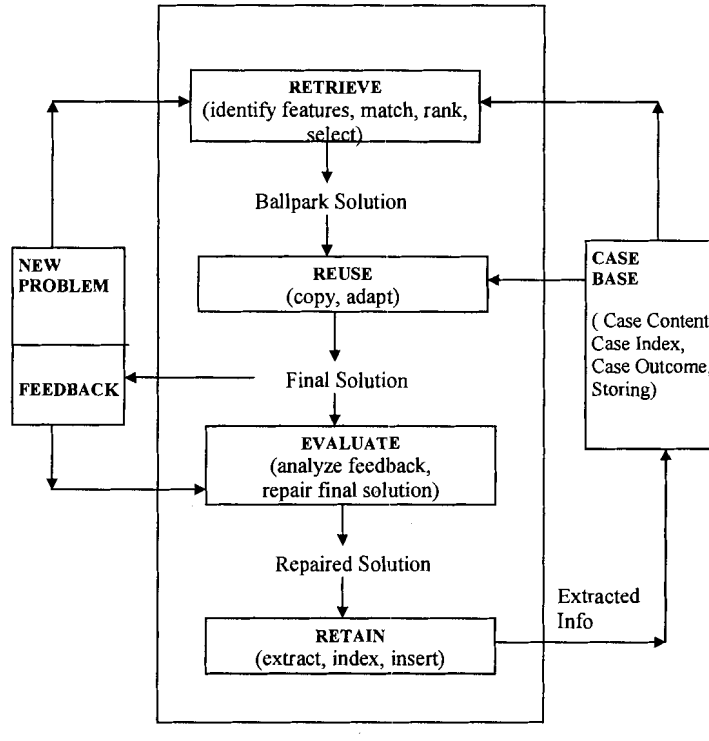

Figure 8.1: *Case-Based Reasoning Process.*

As is clear from the above description, the phases Retrieve and Reuse form the pre-solution phase that computes a solution for the given problem; the phases Evaluate and Retain form the post-solution phase that performs supplementary tasks once the solution has been proposed and its feedback received. Therefore, for modularizing the above functional units, we decompose the working of the case-based reasoner into two modules. The first is the case-based reasoner itself that handles the functionality of retrieve and reuse. The second module is the *Post Solution Module*, that handles the functionality of evaluate and retain. The Post Solution Module is a component of the CBR, and is invoked by the CBR once a solution has been proposed. The main idea behind this break-down is the fact that once a

solution has been proposed, the Post Solution Module can work independent of the CBR; in particular it may take months or years for some feedbacks to be made available (e.g. treating a cancer patient) and under these circumstances the working of the CBR should not stop.

Based on the above description, we come up with basic data structures used in the formulation of CBR (Spec 8.1). The CBR and Post Solution Module are represented by ASM agents that have an associated program, and the Post Solution Module operates asynchronously w.r.t the CBR.

```
// ———————— Basic CBR Definitions ————————
// Domains
domain CBR
domain POST_SOL_MODULE
domain CASE
domain PROBLEM
domain SOLUTION
AGENT ≡ POST_SOL_MODULE ∪ CBR  // ASM Agent

// Main CBR Functions
caseBase : CBR → CASE − Set
postSolModule : CBR → POST_SOL_MODULE
ballParkSolution : CBR → SOLUTION
finalSolution : CBR → SOLUTION
problem : CBR → PROBLEM
```

Spec 8.1: *Basic Definitions for Abstract CBR.*

Based on the above algebraic specification, we can show the CBR process to be as depicted in Spec 8.2. The associated CBR phases are represented by abstract ASM rules viz: RETRIEVE and REUSE. We split the working of the CBR into four modes, collectively called $cbrMode$ — in the **idle** mode, the signaling mechanism is used to trigger the fact that a new problem needs to be solved and the mode then changes to *retrieve*; **retrieve** mode calls the abstract function RETRIEVE, which sets the *ballparkSolution*, and changes the mode to *reuse*; **reuse** mode calls the abstract function REUSE, which sets the $finalSolution$ and changes the mode to *done*; the **done** mode signals the user (system) that the problem has been solved along with providing the final solution, and also triggers the associated Post Solution Module, which once signaled starts working independently.

```
// ———————— CBR Program ————————
// Definitions
domain CBR_MODE ≡ {idle, retrieve, reuse, done}
cbrMode : CBR → CBR_MODE
// Rule
CBRProgram ≡
  case cbrMode of
    idle →
      onsignal s : NEW_PROBLEM
        problem(self) := problem(s)
        cbrMode := retrieve
    retrieve →
      // This sets the ballParkSolution
      // This sets the cbrMode to reuse
      RETRIEVE(self)
    reuse →
      // This sets the finalSolution
      // This sets the cbrMode to done
      REUSE(self)
    done →
      // trigger the PostSolModule to start running in parallel.
      trigger s : INIT, postSolModule(self)
      // Send the solution back to the user.
      trigger s : PROBLEM_SOLVED, user(self)
        solution(s) := finalSolution(self)
      cbrMmode := idle
```

Spec 8.2: *Abstract CBR Program*

Next, we refine the abstract definition of the domain CASE and the rules RETRIEVE and REUSE.

Case Representation

A case-based reasoner has a *case-base*, which is the repository or library of all past experiences or *cases*. A *case* can be defined as a piece of knowledge, that records past experiences of the reasoner in a given context, and also decides the future behavior of the reasoner based on the knowledge it contains.

Abstractly speaking, a case should have the following three constituents: (1) *Problem Descriptors*: the state of the world when the episode recorded in the case occurred. This can

be also be seen as the *index vocabulary* of the case. (2) *Solution*: the solution to the problem the case stores, or the method by which the solution was constructed. (3) *Outcome*: the resulting state of the world after the solution was applied to the problem. This usually records the success or failure of the proposed solution based on the external feedback provided by the user.

Based on the above information, we can now refine the abstract domain CASE to include these three constituents (Spec 8.3).

```
// ———————— Case Representation ————————
domain CASE-Index
domain CASE-Content
domain CASE-Outcome

// Function on CASE
caseIndex : CASE → CASE-Index
caseContent : CASE → CASE-Content
caseOutcome : CASE → CASE-Outcome
```

Spec 8.3: *Refining the Abstract Domain CASE*

The organization and indexing of the case library is crucial as it determines the retrieval of the right case at the right time.

The indexing problem is the problem of making sure that a case is assigned appropriate labels, so that the right case is retrieved at the right time. Deciding on an index vocabulary is a crucial task. An index decides on the usefulness of a case in a given context. Indexes can represent surface features or abstract derived features.

The contents of the case can be organized as an attribute-value (index-value) pair, as a hierarchy of part-subpart relationships or other sophisticated mechanisms may be used.

As the case library becomes large, efficient retrieval becomes a bottleneck and storage mechanisms of hash tables, multi-level index trees etc may be employed.

Clearly, all the aforementioned issues are implementation details that need to be given due thought at the instantiation stage of the abstract reasoner, and thus not further discussed here. We address some of these issues while instantiating a concrete reasoner in Section 8.2.3.

Retrieve

In the abstract CBR, the retrieve phase was represented by an abstract ASM rule called RETRIEVE. We now refine this abstract rule to the next level of detail, although it is still based on abstract functions (Spec 8.4). The overall responsibility of this phase is to find cases in the case base that are similar to the current problem and return a potential solution, called *ballpark solution.* In this light, the retrieval process can be further broken down into sub-tasks, and each sub-task is represented by a mode.

```
// ———————— RETRIEVE ————————
// Definitions
domain RETRIEVE_MODE ≡ {idle, identify, match, rank}
retrieveMode : CBR → RETRIEVE_MODE
identifiedIndex : CBR → CASE-Index
matchedCases : CBR → CASE − Set
// Rule
RETRIEVE(self : CBR) ≡
  case retrieveMode of
    idle →
      retrieveMode := identify
    identify →
      // This sets identifiedIndex
      IDENTIFY(self, problem(self))
      retrieveMode := match
    match →
      // This sets matchedCases, based on identifiedIndex
      MATCH(self, identifiedIndex(self))
      retrieveMode := rank
    rank →
      // This sets the ballparkSolution, based on matchedCases.
      RANK(self, matchedCases(self))
      retrieveMode := idle
      cbrMode := reuse
```

Spec 8.4: *Refining the Abstract Function RETRIEVE.*

The **idle** mode simply enables the *identify* mode. The **identify** mode calls the abstract function IDENTIFY, that identifies the relevant indexes (features/descriptors) of the problem description and sets *identifiedIndex*; the mode is then changed to *match.*

The **match** mode calls the abstract function MATCH that matches the identified indexes (*identifiedIndexes*) of the current problem against the indexes of the existing cases (*caseIndex*) in the case base and sets *matchedCases*; the mode is then changed to *rank*. The match can be an exact match or a partial match. Finally, the **rank** mode calls the abstract function RANK, that uses some pre-defined metric to rank the matched cases (*matchedCases*) and proposes the case with the highest rank as a ballpark solution (*ballparkSolution*). This mode also changes the *cbrMode* to *reuse* and the *retrieveMode* to *idle*; this thus implies that the retrieve phase has completed and the reuse phase should be invoked.

Reuse

We now refine the abstract function REUSE to the next level of detail; albeit still abstract. The responsibility of the reuse function is to return the final solution, based on the ballpark solution (Spec 8.5). This consists of either re-using the ballpark solution as is or adapting it to better suit the current problem. We achieve this by associating three explicit modes with the REUSE function — **idle** mode checks the predicate *isCopy* to determine whether the ballpark solution (*ballparkSolution*) can be copied as is or needs to be adapted, and consequently changes the mode to either *copy* or *adapt*. The **copy** mode simply returns the ballpark solution as the final solution (*finalSolution*). The **adapt** mode uses an abstract function ADAPT to modify the ballpark solution into a final solution. Both the modes, *copy* and *adapt*, change the *cbrMode* to *done* and the *reuseMode* to *idle*, implying that the Reuse process is over and that the CBR can proceed with the next phase.

```
// ———————— REUSE ————————
// Definitions
domain REUSE_MODE ≡ {idle, copy, adapt}
reuseMode : CBR → REUSE_MODE
isCopy : CBR → BOOLEAN
// Rule
REUSE(self : CBR) ≡
  case reuseMode of
    idle →
      if isCopy(self) then
        reuseMode := copy
      else
        reuseMode := adapt
    copy →
      finalSolution(self) := ballparkSolution(self)
      reuseMode := idle
      cbrMode := done
    adapt →
      // This sets finalSolution, by adapting the ballpark Solution
      ADAPT(self, ballparkSolution(self))
      reuseMode := idle
      cbrMode := done
```

Spec 8.5: *Refining the Abstract Function REUSE.*

Post Solution Module

The Post Solution Module is a component of the CBR that handles the Evaluate and Retain phases of the CBR Process. It is defined as an ASM agent (POST_SOL_MODULE), which once triggered by the CBR works asynchronously with respect to the CBR. The overall working of the Post Solution Module includes recording the case proposed as a solution as an *unevaluated case*, and waiting for the external environment to provide the *feedback* for this unevaluated case. Based on the feedback received, the Post Solution Module then analyzes and repairs the unevaluated solution; this is handled by the Evaluate phase. Once the unevaluated solution is repaired, the case base is then updated with this new repaired solution and other *extracted information* that is derived from the external feedback; this is handled by the Retain phase.

Based on the above description, we come up with the basic definitions of the Post Solution

Module (Spec 8.6).

```
// ———————— Definitions for POST_SOL_MODULE ————————
domain FEEDBACK
domain UNEVAL_CASE

// Functions on PostSolModule
parentCBR : POST_SOL_MODULE → CBR
unevalCaseSet : POST_SOL_MODULE → UNEVAL_CASE – Set
// Auxiliary Functions
unevalCase : POST_SOL_MODULE → UNEVAL_CASE
addAsUnevalCase : POST_SOL_MODULE → BOOLEAN
integrateFeedback : POST_SOL_MODULE → BOOLEAN
```

Spec 8.6: *Basic Definitions for POST_SOL_MODULE.*

The abstract domain UNEVAL_CASE represents a case that has been proposed as a solution but for which the feedback is not yet available, and the function *unevalCaseSet* represents a set of all such cases. UNEVAL_CASE can be refined to hold basic information that is required for storing it and later retrieving it, like the associated *problem*, the *proposed solution*, associated *feedback*, the repaired solution derived by repairing the proposed solution based on feedback, the new *information extracted* from the feedback (Spec 8.7).

```
// ———————— UNEVAL_CASE ————————
domain EXTRACTED_INFO
unevalProblem : UNEVAL_CASE → PROBLEM
unevalSolution : UNEVAL_CASE → SOLUTION
feedback : UNEVAL_CASE → FEEDBACK
repairedSolution : UNEVAL_CASE → SOLUTION
extractedInfo : UNEVAL_CASE → EXTRACTED_INFO
```

Spec 8.7: *Refinement of UNEVAL_CASE.*

Based on the above definitions, the working of the POST_SOL_MODULE as given by its Program is depicted in Spec 8.8[1]. The associated phases are represented by abstract ASM

[1]Only relevant portions are shown here.

rules viz: EVALUATE and RETAIN. The process is a sequence of four steps, represented by modes.

```
// ———————— Post Solution Module Program ————————
// Definitions
domain POST_SOL_MODE ≡ {idle, evaluate, retain, done}
psMode : POST_SOL_MODULE → POST_SOL_MODE

// Rule
POST_SOL_MODULE_Program ≡
  case psMode of
    idle →
      onsignal s : INIT
        if addAsUnevalCase(self) then
          extend UNEVAL_CASE with newcase
            // Fill associated info in the new case.
            add newCase to unevalCaseSet(self)
      onsignal s : FEEDBACK_AVAILABLE
        choose x in unevalCaseSet(self) with match(x, s)
          // Sets unevalCase with x and its feedback from environment.
        if none
          extend UNEVAL_CASE with unevalCase
            // Fill associated info into unevalCase.
            add unevalCase to unevalCaseSet(self)
        if integrateFeedback(s) then
          psMode := evaluate
        else
          psMode := done
    evaluate →
      EVALUATE(self)  // Sets repairedSolution, psMode to Retain
    retain →
      RETAIN(self)  // Sets the psMode to done.
    done →
      remove unevalCase(self) from unevalCaseSet(self)
      psMode := idle
```

Spec 8.8: *POST_SOL_MODULE Program.*

In the **idle** mode, the module checks for two kinds of signals. The signal INIT triggered by the associated CBR signals the fact that a solution has been proposed for the given problem, upon which the module calls *addAsUnevalCase*, specified as an abstract derived function, which determines whether the proposed case gets stored as an unevaluated case along with

all associated information, or not. The signal FEEDBACK_AVAILABLE is triggered by the external environment (via the CBR) when the feedback for an existing unevaluated case or a new feedback is available. Upon this signal the corresponding case is then retrieved from the *unevalCaseSet* and its feedback recorded; if there is no match for such a case, i.e the feedback being reported is an independent one, a new *unevalCase* is created with all associated information and stored in the *unevalCaseSet*. Now, the abstract derived function *integrateFeedback* is checked to determine whether to proceed with integrating this feedback into the case-base; if yes, the mode changes to *evaluate*; if no, the mode changes to *done*[2]. The **evaluate** mode calls the abstract function EVALUATE to repair the unevaluated solution based on feedback, which sets *repairedSolution* and changes the mode to *retain*. The **retain** mode calls the abstract function RETAIN, which extracts new information from the feedback by setting $extractedInfo$. It then integrates the repaired solution and the extracted information into the existing case base, and changes the mode to *done*. The **done** mode removes this unevaluated case from the *unevalCaseSet*, since this case has now been successfully evaluated, and changes the mode back to *idle*.

Evaluate

We now refine the abstract function EVALUATE as shown in Spec 8.9. The overall responsibility of the evaluate function is to *analyze* the feedback provided by the external environment and *repair* the unevaluated solution if required. This is achieved by associating four modes with the EVALUATE function, collectively called *evaluateMode*. The **idle** mode simply changes the mode to *analyze*. The **analyze** mode calls the abstract function ANALYZE, which examines the feedback and determines whether repair is indeed needed, sets the *repairNeeded* predicate accordingly, and changes the mode to *check*. The **check** mode checks the *repairNeeded* predicate and changes the mode to *repair* if true. If false, the unevaluated solution (*unevalSolution*) is returned as the repaired solution (*repaired-Solution*); the mode is changed back to *idle*, and *cbrMode* is set to *retain* to invoke the next phase Retain. The **repair** mode calls the abstract function REPAIR, which repairs

[2]Generally, in the CBR life cycle, both the predicates *addAsUnevalCase* and *integrateFeedback* are true, meaning that the proposed solution is always stored as an unevaluated case, and its feedback, when available, is integrated into the case-base. However, in the spirit of being abstract, we give the designer the flexibility to do otherwise.

the solution and updates *unevalSolution* with repaired solution. It then changes the mode to *analyze*, which determines if the repaired result is satisfactory and performs recursive repairing until an acceptable result is reached.

```
// ———————— EVALUATE ————————
// Definitions
domain EVALUATE_MODE ≡ {idle, analyze, check, repair}
evaluateMode : POST_SOL_MODULE → EVALUATE_MODE
repairedNeeded : POST_SOL_MODULE → BOOLEAN

// Rule
EVALUATE(self : POST_SOL_MODULE) ≡
  case evaluateMode of
    idle →
      evaluateMode := analyze
    analyze →
      ANALYZE(self, unevalCase)  // Sets repairNeeded predicate.
      evaluateMode := check
    check →
      if repairNeeded(self) then
          evaluateMode := repair
      else
          repairedSolution(unevalCase) := unevalSolution(unevalCase)
          evaluateMode := idle
          mode := retain
    repair →
      REPAIR(self, unevalCase)  // Updates the unevalSolution(unevalCase).
      evaluateMode := analyze  // Recursively repair sol, until satisfied.
```

Spec 8.9: *Refining the Abstract Function EVALUATE.*

The Evaluate phase ensures that the reasoner is able to evaluate its performance and become more efficient with time.

Retain

We now refine the abstract function RETAIN used in the POST_SOL_MODULE. This phase incorporates new knowledge available from the problem-solving episode into the case base. In particular, by storing the outcome (success or failures) of the proposed solutions, it can lead to positive or negative reinforcement learning. Every time a problem is solved, the case

base is updated. This step also leads to *Learning* in a case-based reasoner.
We do so by associating three modes with this function. The **idle** mode simply enables the *extract* mode. The **extract** mode calls the abstract function EXTRACT, that decides which information from the unevaluated case and feedback to retain, and in what form to retain it; the mode is then changed to *integrate*. The **integrate** mode then calls the abstract function INTEGRATE to integrate the extracted information into the case base. Integration can be done by either inserting a new case, or by updating the existing cases. It then changes the mode back to *idle* and the *cbrMode* to *done*.

```
// ———————— RETAIN ————————
// Definitions
domain RETAIN_MODE ≡ {idle, extract, integrate}
retainMode : POST_SOL_MODULE → RETAIN_MODE
// Rule
RETAIN(self : POST_SOL_MODULE) ≡
  case retainMode of
    idle →
      retainMode := extract
    extract →
      EXTRACT(self, unevalCase)  // Sets extractedInfo.
      retainMode := integrate
    integrate →
      // Integrates the extracted to the CaseBase.
      INTEGRATE(self, extractedInfo(unevalCase))
      retainMode := idle
      cbrMode := done
```

Spec 8.10: *Refining the Abstract Function RETAIN.*

This completes the formalization of the Abstract Case-Based Reasoner. The next step in instantiating a Concrete Reasoner is to refine the abstract functions: IDENTIFY, MATCH and RANK in the function RETRIEVE; ADAPT in function REUSE; ANALYZE, REPAIR in the abstract function EVALUATE; EXTRACT and INTEGRATE in function RETAIN. These functions are typically algorithmic in nature and are determined by the nature and need of the application domain. There exist sophisticated ranking algorithms, adaptation algorithms etc that increase speed and efficiency [67], [82]. We produce simple refinements of these function in Section 8.2.3 to instantiate a concrete reasoner for the path planning problem of our application.

8.2.3 Instantiation of Abstract CBR: Concrete CBR of SEM

In this section we use the abstract CBR as developed in the preceding sections and apply data refinements to the abstract functions and abstract rules, to derive a concrete reasoner for the navigation problem of the SEM. The resultant CBR is called the SEM_CBR and forms the CBR component of the Space Evolution Module (SEM).
Such a step-by-step refinement of the abstract CBR into the concrete, also proves the soundness and generality of the abstract CBR, and also shows the feasibility and the tractability of achieving such a concrete model through refinements.

The SEM_CBR takes a problem that contains a source and destination, and the overall responsibility of the CBR is to return the best path it contains in its case base from the given source to the destination. This path is preferably a complete path, or if such a path cannot be found, a partial path toward the destination. The SEM_CBR does not perform any adaptation, and its post solution module does not evaluate and repair the proposed solution, since these two tasks are not required in the intuitive logic of our problem domain. Every time the SEM is in the *pathPlanning* mode, a call is made to the CBR giving it a problem to solve, and once the path has been traversed, a feedback for the traversed path is made available to the CBR from the SEM mode *pathCompleted.* New information is extracted from the feedback and integrated into the case base[3] This ensures that the CBR becomes more efficient and effective with time.

SEM_CBR is an instantiation of the abstract CBR, that refines its abstract functions and rules followed by the keyword *where*. It is composed of a concrete post solution module SEM_POST_SOL_MODULE, that is an instantiation of its abstract counterpart POST_SOL_MODULE. SEM_POST_SOL_MODULE refines the abstract functions and rules of the abstract POST_SOL_MODULE, followed by the keyword *where*. Spec 8.11 shows some miscellaneous data structures that are refined. The abstract domain SOLUTION is

[3]However, not every problem is marked as an unevaluated problem whose feedback is awaited and not every feedback that is made available is inserted into the case base. In case of a person abandoning the current destination without completing it, an empty feedback is sent by the SEM function Send_Feedback_To_CBR, which is not integrated with the case base. This is checked by the function *integrateFeedback*. In case of the influence factor weights being changed, although a new problem is sent, it is not inserted as an unevaluated case since the problem is still the same of going from the same source to the same destination which already exists as an unevaluated case. This is determined by the function *addAsUnevalCase* which checks if a similar problem already exists in the set of unevalCase or not.

refined to a PATH which is a sequence of Nodes, and the owner of the CBR is refined to be the space evolution module SEM.

```
// ———————— CONCRETE DOMAINS ————————
domain SEM_CBR ≡ CBR  where...
domain SEM_POST_SOL_MODULE ≡ POST_SOL_MODULE  where...

domain PATH ≡ NODE − Seq
SOLUTION ≡ PATH
OWNER ≡ SEM
```

Spec 8.11: *Concrete CBR and POST_SOL_MODULE.*

The abstract domain CASE holds the information for the CASE-Index, CASE-Content and CASE-Outcome, which are refined as shown in Spec 8.12.

```
// ———————— CONCRETE CASE ————————
// —CASE-Index—
source : CASE-Index → NODE
dest : CASE-Index → NODE
timeType : CASE-Index → TIMETYPE
date : CASE-Index → DATE

// —CASE-Content—
path : CASE-Content → PATH

// —CASE-Outcome—
frequency : CASE-Outcome → VALUE
reinforcement : CASE-Outcome → REINFORCEMENT
tripImportance : CASE-Outcome → TRIP-IMPORTANCE
```

Spec 8.12: *Refining the Abstract Domain CASE.*

The abstract domain PROBLEM holds all the information about the problem posed to the CBR. It holds the source and destination node of the path to be computed ($source, dest$), the time when this problem was given to the CBR ($time$), and the parameter *closeness* which decides by how may hops the partial path can be away from the final destination.

The abstract domain FEEDBACK stores the actual path taken by the person (*takenPath*). The two domains are formulated in Spec 8.13.

```
// ---------- PROBLEM ----------
source : PROBLEM → NODE
dest : PROBLEM → NODE
time : PROBLEM → TIME
closeness : PROBLEM → INTEGER

// ---------- FEEDBACK ----------
pathTaken : FEEDBACK → PATH
```

Spec 8.13: *Refining the Abstract Domain PROBLEM and FEEDBACK.*

Next, we refine the abstract rules. We start with the abstract CBR rules, and then move on to POST_SOL_MODULE rules.

The abstract rule IDENTIFY is responsible for identifying the problem descriptors from the given problem (*problem*). These descriptors then help in retrieving the appropriate cases, by matching them against the case indexes. The task of identifying these descriptors is simple, as the structure of the problem already contains them; the contents of the $identifiedIndex$ (source, dest etc.) are then merely copied from the *problem* (Spec 8.14).

```
// ---------- IDENTIFY ----------
IDENTIFY(self : CBR, problem : PROBLEM) ≡
  source(identifiedIndex) := source(problem)
  dest(identifiedIndex) := dest(problem)
  timeType(identifiedIndex) := timeType(time(problem))
  date(identifiedIndex) := date(time(problem))

where
  identifiedIndex ≡ identifiedIndex(self)
```

Spec 8.14: *Refining the Abstract Rule IDENTIFY.*

The abstract rule MATCH, refined in Spec 8.15 is responsible for matching the identified indexes against the indexes of the cases, and storing all those cases that fulfill the matching

criteria as matched cases *matchedCases*. This is basically a two step process. First, the case base is checked for cases that yield an exact match, i.e cases that store complete paths are retrieved. The criteria for an exact match as performed by the derived function $exactMatch$ is that the source and destination of the identified index ($identifiedIndex$) are the same as the source and destination of the case index($caseIndex(c)$). All such cases are added to the set of matched cases. However, if no exact matches are found, a search is made for partial paths, i.e a path whose destination is 'close enough' to the final destination. The parameter *closeness* gives the maximum number of hops by which the destination of the partial path can be away from the final destination. The case base is now searched for such cases, starting with the closeness of 1 and monotonically incrementing the number of hops; as soon as matched cases are found for a given level of hop, the next increment of the hop is not made, i.e preference is given to paths that are closer to the final destination. The function $partialMatch$ performs a check for partial paths by comparing the size of the best path from the tail node of the retrieved path to the final destination, against the set number of hops, such that the size is either less than or equal to the number of hops.

```
// ———————— MATCH ————————
MATCH(self : CBR, identifiedIndex : CASE-Index) ≡
  forall c ∈ caseBase(self) with exactMatch(c, identifiedIndex)
     add c to matchedCases(self)
  seq
  if matchedCases(self) = {} then
    let hops = 1 in
     while matchedCases(self) = {} ∧ hops < closeness(problem(self))
       forall c ∈ caseBase(self) with partialMatch(c, identifiedIndex, hops)
         add c to matchedCases(self)
       hops = hops + 1

where
  exactMatch(c, identifiedIndex) ≡ source(caseIndex(c)) = source(identifiedIndex)
                                 ∧ dest(caseIndex(c)) = dest(identifiedIndex)
  partialMatch(c, identifiedIndex, hops) ≡ source(caseIndex(c)) = source(identifiedIndex)
            ∧ Size(bestPath(dest(caseIndex(c)), dest(identifiedIndex))) <= hops
```

Spec 8.15: *Refining the Abstract Rule MATCH.*

The abstract rule RANK ranks all the cases is the set of matched cases and returns the one

with the highest rank as the ballpark solution *ballparkSolution*. The ranking is executed by first sub ranking the set of matched cases (*matchedCases*) by the function DoSubRank, and then selecting the case with the latest date (*highestDate*) as the ballpark solution. The

```
// ---------- RANK ----------
RANK(self : CBR, matchedCases : CASE − Set) ≡
  DoSubRank(self)
  seq
  choose c in subrankedCases(self) with highestDate(c)
    ballparkSolution(self) := path(caseContent(c))
where
  highestDate(c) ≡ ∀x(x ∈ subrankedCases) ⇒ date(caseIndex(c)) ≥ date(caseIndex(x))

DoSubRank(self : CBR) ≡
  forall c in positiveCases with highestSubRank(c)
    add c to subrankedCases(self)
where
  highestSubRank(c) ≡ ∀x(x ∈ positiveCases()) ⇒ subRank(c) ≥ subRank(x)
  positiveCases() ≡ {all n|n ∈ matchedCases(self)
                     ∧ reinforcement(caseOutcome(c)) ≠ negative)}
  subRank(z) ≡ weightCost(self) * CostValue(z)
             + weightOutcome(self) * outcomeValue(z)
             + weightTime(self) * rightTime(z)
```

Spec 8.16: *Refining the Abstract Rule RANK.*

DoSubRank function performs sub ranking only on those set of matched cases that have a positive outcome, i.e those cases that have a reinforcement or experience associated with them that is not negative. This thus incorporates both *positive and negative reinforcement learning* in our navigation algorithm. A sub rank is then assigned to each of the positive cases, which is a weighted sum of *costValue*, *outcomeValue* and *rightTime*. Cost Value is the summation of perception attributes PER_EDGE_ATTR (distance, road type, traffic etc.) of the constituent edges of the path, as given by the SUBJ_ENV of the person. A point to be noted is that in calculating the costValue of the path, the current real-time local information of the edges adjacent to the person's current position are considered. Outcome Value can typically be the summation of the activity attributes AC_EDGE_ATTR (frequency, reinforcement, trip importance) of the constituent edges, as given by the SUBJ_ENV of the person; or it can be derived from AC_EDGE_ATTR values stored in the CASE-outcome,

which are the values of the path as a whole. Right Time is a boolean value that returns a true if the case path was taken in the same time interval (morning, afternoon, evening) as the time interval of the problem, or 0 otherwise. Tacit representation of RANK is given in Spec 8.16, for details of function *costValue*, *outcomeValue*, *rightTime*, refer to AppendixA.

We do not perform any adaptation of the ballpark solution in the SEM_CBR. This is achieved by the derived function *isCopy* that returns a value true.

Next, we refine the abstract function and rules of the abstract post solution module, POST_SOL_MODULE. The concrete post solution module is called SEM_POST_SOL_ MODULE. The abstract functions *addAsUnevalCase* determines whether the new problem posed to the CBR gets stored as an unevaluated case whose feedback is then awaited. In case of the influence factor weights being changed, although a new problem is sent, it is not inserted as an unevaluated case since the problem is still the same of going from the same source to the same destination, which already exists as an unevaluated case. *addAsUnevalCase* performs this check (Spec 8.17).
integrateFeedback determines whether the feedback that is made available for an unevaluated problem is integrated into the case base. In case of a person abandoning the current destination without completing it, an empty feedback is sent by the SEM function Send_Feedback_To_CBR, which is not integrated with the case base. This is checked by the function *integrateFeedback* (Spec 8.17).

```
// ---------- addAsUnevalCase ----------
addAsUnevalCase(self : POST_SOL_MODULE)
  choose x ∈ unevalCaseSet(self) with unevalProblem(x) = problem(parentCBR(self))
                                   ∧ unevalSolution(x) = solution(parentCBR(self))
      return true
  if none
      return false
// ---------- integrateFeedback ----------
integrateFeedback(fa : FEEDBACK_AVAILABLE)
  if takenPath(externalFeedback(fa)) <> [] then
     return true
  else
     return false
```

Spec 8.17: *addAsUnevalCase and integrateFeedback.*

The abstract rule ANALYZE which decides whether the proposed solution needs repair or not, by setting the predicate *repairNeeded*, sets it to false in the SEM_POST_SOL_MODULE.

The abstract rule EXTRACT extracts all the information that is made available from the external feedback stored in the *unevalCase*, which basically is the path that the person has taken. In order to extract all such information, the domain EXTRACTED_INFO is refined to be a set of CASEs, so that it makes the integration of information into the case base easier. From the path that the person traversed, a combination of other paths can be derived, i.e we can compute its transitive closure; for e.g if a person went from A-B-C, it also 'knows' the paths B-C, C-B-A . We calculate all such permutations using the function GetAllPermutations, which returns a set of new paths *newPathSet*. For each of these paths, we then extract the indexes, content and outcome using the functions Extract_Index, Extract_Content and Extract_Outcome respec. These functions perform simplistic work and can be found in Appendix B. Each of these extracted cases is then added to the set of extracted info. It should be noted that while we compute all possible permutations of the taken path, we also have the option of computing all new paths created by concatenating two paths that gives a path which hasn't been directly traveled on, but whose sub-paths have been traveled on. This however produces sub-standard results as it can return paths that are clearly not desirable and unintuitive, and hence we do not employ this technique.

```
// ---------- EXTRACT ----------
EXTRACTED_INFO ≡ CASE − Set

EXTRACT(self : POST_SOL_MODULE, unevalCase : UNEVAL_CASE) ≡
  let newPathSet(self) ← GetAllPermutations(pathTaken(feedback(unevalCase))) in
  // set of all possible paths.
  forall path in newPathSet(self)
    extend CASE with extractedCase
      caseIndex(extractedCase) := Extract_Index(path)
      caseContent(extractedCase) := Extract_Content(path)
      caseOutcome(extractedCase) := Extract_Outcome(path)
      add extractedCase to extractedInfo(unevalcase(self))
```

Spec 8.18: *Refining the Abstract Rule EXTRACT.*

Finally, we refine the abstract rule INTEGRATE in Spec 8.19. This rule is responsible for integrating all the extracted information *extractedInfo* into the case base. This can be

done in two ways, either by inserting a new case or by updating an existing case with the new information. For each extracted case in the set of extracted cases, a check is performed to determine if this case already exists in the case base. This is done by the derived function *matchExists* that returns true if the source, destination, path, and time type of the case in the case base are all the same as the new extracted case. In this case the function UpdateCase is called to merely reset or update the relevant information of this existing case to incorporate new extracted information. However, if no such case exists, then the extracted case is inserted into the CBR as a new case.

```
// ———————— INTEGRATE ————————
INTEGRATE(self : POST_SOL_MODULE, extractedInfo : CASE − Set) ≡
  forall newCase in extractedInfo(unevalCase(self))
     choose oldCase from caseBase with matchExists(oldCase, newCase)
        UpdateCase(oldCase, newCase)
     if none
        add newCase to caseBase

  where
  matchExists(oldCase, newCase) ≡
                source(caseIndex(oldCase)) = source(caseIndex(newCase))∧
                dest(caseIndex(oldCase)) = dest(caseIndex(newCase))∧
                timeType(caseIndex(oldCase)) = timeType(caseIndex(newCase))∧
                path(caseContent(oldCase)) = path(caseContent(newCase))
  caseBase ≡ caseBase(parentCBR(self))
```

Spec 8.19: *Refining the Abstract Rule INTEGRATE.*

Chapter 9

Shortest Path Planning

The problem of Navigation is a complicated one, that in simple terms can be viewed as moving an entity from source S to destination D by first deciding the different paths that can be taken, evaluating the cost of taking these paths under given circumstances, choosing the shortest (most suitable) path, and finally executing this theoretical route by actually moving the entity.
Taking the shortest path or the path with least cost is a sub-problem of this bigger problem. This chapter is devoted to analyzing shortest path algorithms and deriving one as per the demands of our application. The algorithm presented here forms the Model-Based Reasoning (MBR) component of the Space Evolution Module.
We survey some basic algorithms in Section 9.1 and present our approach in Section 9.2.

9.1 Shortest Path Problem

The shortest path problem is the problem of finding a path from source S to destination D, that has the least cost associated with it, where the cost parameters can be anything such as distance, travel time, etc. Formally, this problem can be stated as follows [29]:
In a shortest-path problem, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow R$, mapping edges to real-valued-weights. The weight of path

$p =< v_0, v_1, ..., v_k >$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{9.1}$$

The shortest-path weight from u to v is defined by:

$$\delta(u,v) = \begin{cases} \{w(p) : u \xrightarrow{p} v\} & if\ there\ is\ a\ path\ from\ u\ to\ v. \\ \infty & otherwise. \end{cases} \tag{9.2}$$

A shortest path from vertex u to v is then defined as any path p with weight $w(p) = \delta(u,v)$.

Calculating shortest paths for a given graph can be discerned into the following categories: one-to-one, one-to-some, one-to-all (single source shortest path), all-to-one, all-to-all (all source shortest path).

9.1.1 Shortest Path Algorithms

The field of shortest path algorithms is well-studied and entails decades of research and experimentation. The shortest path problem is also one of the most fundamental Network Optimization as well as Intelligent Transportation problems.

In [33] is a description of the most famous shortest path algorithm viz Dijkstra's Shortest Path Algorithm. This *greedy* algorithm guarantees to find the optimal shortest path in the given graph with non-negative edges in time $O(n*n)$. There are several implementations of this basic algorithm to improve time and space efficiency, using sophisticated data structures of heaps, queues, buckets etc. Most other shortest path algorithms are variations of this generic algorithm. A good description of the classical algorithms and their implementation appears in [49].

There is a class of shortest path algorithms that use a form of *Heuristic Search*. These set of algorithms such as $A*$ have some estimate (heuristic) of how far from the goal any vertex is. A* is the classical game algorithm that is most widely used in gaming and AI [78]. These algorithms work faster since the use of heuristic avoids looking in directions with fruitless search. The construction of this heuristic function involves some overhead and should be

weighed against the yielded benefits. Although these algorithms do not guarantee to find the most optimal path, under certain imposed conditions (of the heuristic function), optimality can be achieved. One has to consider the trade-off between speed and optimality while using this class of algorithms.

Liner Programming and Dynamic Programming techniques have also been explored in calculating shortest paths [29]. Another genre of shortest-path algorithms is the *Bi-Directional Search*, which entails computing a path from both origin and destination, and meeting in the middle.

A survey of shortest path algorithms for dynamic graphs can be found in [62]. Path planning is dynamic when the path is continually recomputed as more information becomes available. Such algorithms mainly aim to optimize the creation, updation, and maintenance of the associated data structures affected by graph updates, to ensure time efficient solutions.

Several surveys and experimental evaluations have been carried out to compare the performance of different shortest path algorithms : classical and new.
In [26] , the authors carry an exhaustive study of 17 shortest path algorithms including the Dijkstra's algorithm and its varying implementations. A number of simulated networks with varying degrees of complexity are used. The results of their study suggest that there is no universally best algorithm for all problems; however for graphs with non-negative edges, Dijkstra's algorithm outperforms the rest.
Zhan and Noon [75] test 15 of the 17 shortest path algorithms on 21 real road networks. In their study, Dijkstra-based algorithms outperform other algorithms.
Based primarily on above two studies, Zhan [96] identifies three fastest algorithms for real road networks, two of which are Dijkstra based.

9.2 Our Approach: Proposed Shortest Path Algorithm

This section explains our approach for a shortest path algorithm, that is best suited to the context of our application. This algorithm forms the Model-Based Reasoning component of the Space Evolution Module (SEM). We start by providing an informal explanation of the algorithm is Section 9.2.1, and subsequently the formal specification in ASM syntax in Section 9.2.2.

9.2.1 Overview

The shortest path algorithm that we present reflects natural and intuitive decisions a person makes while moving in an urban landscape. The algorithm depicts the path planning process based on the fact that the person is given a *road map* of the underlying urban landscape, which he uses for making decisions. The path taken might not be a globally optimal and the best one, but it is a more natural and a good-enough one. The algorithm takes into account the factors that are known to influence human path planning.
Typically, the following factors play an integrated role in influencing the path selection process. We name these factors *Influence Factors.*

1. *Distance* - typically distance of travel is sought to be minimized.
2. *Road Type* - people tend to take major roads compared to minor roads.
3. *No. of Intervening Stops* - people tend to take routes with lesser intervening stops.
4. *Angle* - generally people do not travel in the opposite direction of the destination, and thus angle toward the destination is sought to be minimized.
5. *Traffic* - people tend to avoid roads with heavy traffic.
6. *Road Condition* - roads that are under construction, not well-made, dangerous to take are certainly avoided.
7. *Familiarity* - people use familiar roads more often; a road taken once has higher likelihood of being taken again. This corresponds to *behavioral reinforcement learning.*
8. *Quality of Experience* - people tend to take roads with which they associated positive experiences, e.g if one gets robbed on a road, one would try to avoid that road. This corresponds to *positive or negative reinforcement learning.*

Some of these factors are *static* — that typically do not change over time, and some *dynamic* — those that may change over time. The *Influence Factors* are tabulated in Figure 9.1.

As described in Chapter 6, with each edge we associate a set of geographic attributes GEO_EDGE_ATTR , the 'perceived' values of which form the $SUBJ\_EDGE\_ATTR$.

Static Factors	Dynamic Factors
Distance	Traffic Density
No. of intervening Nodes	Road Condition
Angle	Familiarity
Road Type	Experience

Figure 9.1: *Path Influence Factors.*

Thus, the values for the factors *distance, road type, traffic density, road condition* come from GEO_EDGE_ATTR of the GEO_ENV and PER_EDGE_ATTR of the $SUBJ\_ENV$. The values for $familiarity$ are derived from the attributes $frequency$, $tripImportance$ and $intensity$ of Ac_EDGE_ATTR and AW_EDGE_ATTR. The values for *experience* are derived from the attribute $reinforcement$ of the AC_EDGE_ATTR. The values of all these factors are person specific, and represent how different people 'perceive' the same environment. Thus, in this way, we incorporate subjective human perception in calculating their respective preferred paths.
The factors *angle* and *number of intervening nodes* are dependent on the orientation and length of the path and not dependent on edges solely, and hence are computed alongside by the path finding algorithm.

With each factor is associated a *Factor Weight*, which decides the importance of that factor in the overall edge preference. These weights reflect personal preferences and vary from individual to individual. They may change dynamically during the course of travel. For e.g, while starting a journey, minor roads are taken first to get on a highway, and once a highway is taken, we keep traveling on it, and then revert to taking minor roads once closer to the destination; thus the weight for *road type* changes with time.
The overall preference of an edge is then a weighted sum of all the aforementioned factors[1].

When a person has access to a map, he or she has access to general knowledge about the roads such as distance, directional orientation, # of intervening nodes, road type. In other words, for the *static influence factors*, the navigator can easily get information from the map. Generally, a navigator also has some a priori knowledge about the environment from

[1]The factor values are normalized to fall between an interval of 0-1, unless we wish to associate exceptionally high or low preferences to force an agent to take or avoid an edge respc. The factor values are also relative to the length of the edge.

past experiences, such as traffic density, road conditions, familiarity, quality of experience. In other words, for the *dynamic influence factors*, the navigator has partial knowledge about the values based on past experiences, which may not correspond to real-time absolute values. Thus, based on the information made available by opening a map (static attributes), and past knowledge (dynamic attributes), one can easily compute potential routes from any given source to destination. We name the preference of a person for such a path as *Global Path Preference.*

In addition to global path knowledge based on past experiences, a person while moving discovers real-time information about the roads — for e.g. a person might have thought of Sam Street as being a low traffic street, but now while traveling on it, he/she discovers there is a heavy traffic jam on it. Thus, the algorithm should also consider the real-time local information that is discovered by the navigator as and when he/she moves on the chosen path. The idea is to avoid taking roads with dense traffic once discovered on-the-fly, by revising the previous path selection decision. In other words, for the *dynamic influence factors*, the values can be revised to real-time absolute values once you are on that edge. We incorporate this specialized, current knowledge about the roads in the algorithm as the *Local Edge Preference* of an agent for that *edge.*

Hence, our navigation algorithm reflects a balance between *Global Path Planning*, and *Real-Time Local Negotiation.*

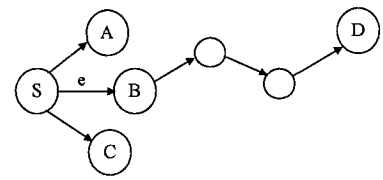

Figure 9.2: Selecting a Path from Source S to Destination D

Assume a person wants to move from source S to destination D (Figure 9.2). The preference of an edge e for a person agent a is defined as:

$$
\begin{aligned}
&edgePreference(e, D) \equiv \\
&\quad localWeight * localEdgePref\,(e, D) \\
&\quad + globalWeight * globalPathPref\,(e, D) \\
&where
\end{aligned}
$$

- $globalPathPref(e, D)$ corresponds to Global Path Preference — preference of taking a 'best' path from B to destination D. We use a Dijkstra like algorithm to compute an all-pairs shortest path. The cost function is based on weighted sum of the influence factor values, where the values of static factors ($GEO\_STAT\_EDGE\_ATTR$) are based on information made available from the map (GEO_ENV), and the dynamic factors ($SUBJ\_EDGE\_ATTR$) are based on past experiences ($SUBJ\_ENV$).

- $localEdgePref(e, D)$ corresponds to Local Edge Preference. It is a weighted sum of all the Influence Factor values. For both the static and dynamic geographic factors (GEO_EDGE_ATTR), the values are based on the current real-time values derived from GEO_ENV as opposed to past learnt values. This gives the algorithm the power to perform local negotiation.

- The effect of *Global Planning* and *Real-Time Local Negotiation* in the overall edge preference is controlled by the weights assigned to each type of preference, namely $globalWeight$ and $localWeight$ respectively. When $globalWeight$ is 0, the algorithm becomes purely 'greedy'; this would be similar to using a random search that may result in back-tracking and fruitless result. When both the weights are set to 1, the algorithm becomes purely global in nature; this would be similar to using Dijkstras',

which guarantees a globally optimal path based on influence factors. A non-zero value of both weights, makes the algorithm exhibits a combination of Global Planning and Local Negotiation.

The algorithm can be seen as a combination of A* [78] and Dijkstra's [33] algorithm for finding the shortest paths. The function *edgePreference*, looks similar to the one used in A* algorithm where *localEdgePref* corresponds to the cost function and *globalEdgePref* corresponds to the heuristic function. However, our algorithm is significantly different from A* since the function *globalEdgePref* is an accurate estimate of the cost based on well-defined influence factors. Also, A* is essentially global in nature, while our algorithm can incorporate combinations of global planning and local negotiation, by playing with the weights of *globalWeight* and *localWeight*.

9.2.2 ASM Specification: Path Explorer Submachine

This section describes the ASM specification of the shortest path planning algorithm described in the previous section. The process is represented by an ASM submachine, and for simplicity we refer to it as *Path Explorer Submachine*, which represents the Model-Based Reasoner (MBR) of the Space Evolution Module (SEM). Spec 9.1 gives the ASM specification of the *PATH_EXPLORER* submachine at the highest level of abstraction.

The submachine has some abstract data structures associated with it, defined in the definitions section of the Spec; logically these are all stored in the volatile memory (see agent architecture). These are refined later on. The submachine takes three parameters viz: the *SEM* (*sem*) that it is a part of, the current node of the person which is the source of the path to be computed (*currentNode*), and the destination of the path (*destNode*).
The function *GLOBAL_RE_CALC* of the submachine is the actual performer of all calculations required to compute a shortest path. The submachine merely uses the data structures set by this function to return the shortest path. The function is detailed later.
The predicate *readyToExplore* is checked to determine whether the function *GLOBAL_RE_CALC* has performed all the calculations or not. If not, the function is called and the predicate is set to true. Once the predicate is true, the submachine 'chooses' an edge in the set of all edges incident to the current node, that has the maximum value (*maxPref*) of the edge preference, *edgePref*.

```
// ———————— Path Explorer Submachine ————————
// Definitions
// Kept in 'Volatile Memory'
edgePref : SEM × EDGE × NODE → PREF_VALUE
readyToExplore : SEM → BOOLEAN
bestPath : SEM × NODE × NODE → PATH

// Rule
GET_SUGGESTED_PATH_Explorer(sem : SEM, currentNode : NODE, destNode : NODE) ≡
  if readyToExplore then
    choose edge in outIncidentEdges(currentNode) with maxPref(edge)
      return concat(currentNode, bestPath(tail(edge), destNode))
  else
    GLOBAL_RE_CALC
    readyToExplore := true
where
  maxPref(edge) ≡ ∀e(e ∈ outIncidentEdges(currentNode)
                  ⇒ edgePref(sem, edge, destNode) ≥ edgePref(sem, e, destNode))
initialize : readyToExplore = false
```

Spec 9.1: *PATH EXPLORER Submachine.*

The preference of an edge is defined using the equation of the previous section, and is represented in the submachine as a derived function $edgePref$ (Spec 9.2).

As described before, $globalWeight(a)$ and $localWeight(a)$ are the agent-specific (a) values of weights for global planning and local negotiation; logically these can be seen as stored in the 'Profile' of the agent.

$globalPathPref$ is a derived function that stores the preference of the *shortest path* from from the source which is the tail of the current edge e, to the *dest*, for an agent a. This preference is based on the factor values derived from agent's past experiences. The values of $globalpathPref$ are set by the $GLOBAL\_RE\_CALC$ function. $globalPathPref$ represents the Global Planning phase of the algorithm.

$localEdgePref$ is also a derived function, that gives the preference of the current edge e, based on real-time current values of the factors. Although to calculate the preference of an edge we don't need the destination, it is specified here as it is used for some other purposes, explained later on. These two derived functions can be seen as stored in the 'Volatile Memory' of the agent, since they re-computed with varying conditions. $localEdgePref$

```
// ———————— edgePref ————————
// Kept in 'Profile'
globalWeight : SEM → WEIGHT_VALUE
localWeight : SEM → WEIGHT_VALUE
// Kept in 'Volatile Memory'
localEdgePref : SEM × EDGE × NODE → PREF_VALUE
globalPathPref : SEM × NODE × NODE → PREF_VALUE

edgePref(a, e, dest) ≡
  globalWeight(a) * globalPathPref(a, tail(e), dest)+
  localWeight(a) * localEdgePref(a, e, dest)
```

Spec 9.2: *edgePref Function.*

represents the Local Negotiation phase of the algorithm.

The submachine finally returns the overall shortest path based on global planning and local negotiation. This is the concatenation of the current node *currentNode* with the path from tail node of e (the chosen edge) to the destination *dest*, as given by *bestPath*. *bestPath* is set by the GLOBAL_RE_CALC function, and represents the path with the highest value of *globalPathPref*.

We now refine the submachine to the next level of detail, to define the derived functions *localEdgePref* and *globalPathPref* used above. In order to do so, we first specify the *Influence Factors* that influence the path selection process (Spec 9.3).

```
domain INDUCED_FACTOR
domain FACTOR ≡ SUBJ_EDGE_ATTR ∪ INDUCED_FACTOR

// Induced Factors
numberOfStops :  → INDUCED_FACTOR
angle :  → INDUCED_FACTOR
// Factor Weights kept in the 'Volatile Memory'
factorWeight : SEM × FACTOR → FACTOR_WEIGHT
```

Spec 9.3: *Influence Factors and Weights.*

The domain *FACTOR* represents all the influence factors collectively. It is composed of the

subjective edge attributes $SUBJ\_EDGE\_ATTR$ which represent the factors distance, road type, road conditions, traffic, familiarity etc; and induced factors $INDUCED\_FACTOR$ that don't belong to an edge but are the properties of the path taken, viz: angle, and # of intervening nodes.
Furthermore, with each factor is associated a *factor weight* which determines the extent of influence of that factor in the overall preference. The factor weights are agent-specific and also dynamic with respect to time. For e.g. if a value 0 is assigned with the factor *roadType*, this would mean the agent doesn't discern between major and minor roads; if the value is changed to non-zero, this would mean the agent has preferences for road types. We associate high preference values with major roads than with minor roads, and this gets reflected in the overall preference value of the edge.

The derived function $localEdgePref$ can now be specified in terms of $localFactorValue$ as given in Spec 9.4. For local edge preference of an edge e, we calculate a weighted sum of all the influence factors. If the factor is a geographic attribute (GEO_EDGE_ATTR), and the edge e is immediately adjacent to an agent's current position, the value of the factor is derived from the *geoEdgeAttr*; meaning that the agent can 'see' the current statistics of the edge and counts on real-time values, rather than relying on values based on past experiences only[2]. If the agent is not adjacent to edge e, or the factor being considered is not a GEO_EDGE_ATTR (but a $SUBJ\_ATTR$), the value of the factor is derived from $subjEdgeAttr$; meaning the agent counts on past experiences only to calculate the preference of this edge. For the value of angle, a triangle composed of head node of edge, tail node of edge, and destination node is formed; meaning the preference of an edge takes into account that the person moves toward the direction of destination, not opposite from it.

The function $globalPathPref$ and $bestPath$ are set by the function $GLOBAL\_RE\_CALC$. $GLOBAL\_RE\_CALC$ performs all-pairs shortest path computation on the given GEO_ENV for all nodes. In computing such a path, the preference value considered for each edge ($gEdgePref$) is based on *geoStaticEdgeAttr* for the $GEO\_STAT\_EDGE\_ATTR$ and on $subjAttr$ for all other attributes. This is because exploring a path is analogous to planning a path by having access to a road map. In such a case, the person can easily see and derive

[2]Here, the terms GEO_EDGE_ATTR and PER_EDGE_ATTR are used interchangeably, since they both have the same constituents. Ideally, the value is not derived solely from *geoEdgeAttr*, but based on an 'interpretation' or 'perception' of this value.

// Local Edge Pref
$localEdgePref(a, e, dest) \equiv \sum_{f \in FACTOR} localFactorValue(a, f, dest, e)$
// Local Factor Value
$localFactorValue$: SEM × FACTOR × NODE × EDGE → FACTOR_VALUE
$localFactorValue(a, f, dest, e) \equiv$

$$\begin{cases} angle(dest, e) * factorWeight(a, f) & : f = angle \\ 1 & : f = numberOfStops \\ interpret(geoEdgeAttr(e, f)) * factorWeight(a, f) & : f \in \text{GEO\_EDGE\_ATTR} \wedge \\ & \quad [\, currentNode(a) = head(e) \\ & \quad \vee currentNode(a) = tail(e))] \\ subjEdgeAttr(parentAgent(a), e, f) * factorWeight(a, f) & : otherwise. \end{cases}$$

Spec 9.4: *localEdgePref in terms of localFactorValue.*

the values of static geographic attributes (distance, road type etc) from the *GEO_ENV* which are the actual values[3]. For all other attributes however, such as traffic, reinforcement it still has to rely on its past experiences, i.e *SUBJ_ENV*. This can be seen in Spec 9.5, where *globalFactorValue* is a weighted sum of all the factor values.

// Global Edge Pref
$gEdgePref(a, e) \equiv \sum_{f \in FACTOR} globalFactorValue(a, f, e)$
// Global Factor Value
$globalFactorValue$: SEM × FACTOR × EDGE → FACTOR_VALUE
$globalFactorValue(a, f, e) \equiv$

$$\begin{cases} 0 & : f = angle \\ 1 * factorWeight(a, f) & : f = numberOfStops \\ interpret(geoStaticEdgeAttr(e, f)) * factorWeight(a, f) & : f \in \text{GEO\_STAT\_EDGE\_ATTR.} \\ subjEdgeAttr(parentAgent(a), e, f) * factorWeight(a, f) & : otherwise. \end{cases}$$

Spec 9.5: *gEdgePref in terms of globalFactorValue.*

It should be noted that the function *GLOBAL_RE_CALC* is called every time there is a change in the *factor weights*, or *SUBJ_ENV*, since these are the two dynamic data structures it depends on.

[3]Ideally, the value is not derived solely from *geoEdgeAttr*, but based on an 'interpretation' or 'perception' of this value.

We do not show here the specification of $GLOBAL\_RE\_CALC$, since it is only an algorithmic function. The specification can be found in APPENDIX B. It is sufficient to say that the function applies the Dijkstra algorithm for computing an all-pair shortest path. This can typically be replaced by any other algorithm.

The $SUBJ\_ATTR$ values of an edge, i.e the values for factors based on past experience, can be seen as a *case* for that edge. In this manner, we can say the MBR works with *edge cases* represented implicitly as edge attributes, while the CBR works with an explicit case-base that stores path level information. The Path Explorer or the MBR thus makes edge level decisions, while the CBR makes path level decisions. The edge cases are updated dynamically as and when a person traverses an edge or becomes 'aware' of it. This way the MBR is constantly 'learning' and becomes more efficient and effective with time.

In conclusion, a note on the use of hierarchical graphs for path planning is noteworthy. Instead of using a one level 'flat' graph for the computation of shortest path, a *multi-level hierarchical graph* may be used. The main aim of one class of hierarchical graph algorithms is pruning of search space so as to increase the time efficiency. By using a hierarchical graph model, the underlying structure of the complex topographical map can be exploited, that while path calculation may restrict search space to a sub-set of the graph, and thus result in search time and speed benefits. [88], [63] use such a structure and develop algorithms that significantly reduce the search space and time as compared to A*, and also guarantee the optimality of the algorithms.

A second class of hierarchical graph algorithms use such a graph structure with the aim of introducing abstraction in addition to space pruning, i.e computing the shortest path w.r.t a particular level of the hierarchy and then refining it to other levels of the hierarchy [70] [94]. While these approaches introduce the more natural way in which people make path planning decisions, they are sub-optimal, and in some may even greatly compromise on the optimality of the path by forcing people to take non-intuitive paths.

Our approach uses a flat graph, mainly because at this stage we are not concerned with time and speed efficiency, since we are modeling the navigation process at a semantic level. However, one major benefit of such a pruning of search space would embody the fact that people search for paths only in the direction of their destination, i.e it will combine *direction finding* with *route finding*. This has been kept in mind and may be considered as a possible future extension of the underlying graph structure.

PART IV

EXECUTABLE MODEL

Chapter 10

The AsmL Executable Model

This chapter is devoted to deriving an abstract executable semantics of the ASM ground model of the previous sections. The executable model is obtained by applying a refinement step in AsmL — a formal executable specification language — to the ground model.

Executable specifications are important as they provide a means for experimental validation and verification of the system. Since 60-80% of errors are introduced in the requirements engineering phase, rigorous specifications can significantly improve the quality and precision of the modeled system [4]. Executable Specifications (ES) help achieve a high level of confidence in the correctness of the system, by detecting errors in earlier stages of software development, which significantly reduces the repair cost of errors in the later stages of development. Executable specifications can be used for test-case generation, runtime verification, and scenario-based modeling and testing [54].
Furthermore, ES can also form the basis for a simulation model to verify the operation of the modeled part in the wider system context. Behavioral simulations help us analyze system behavior and evolution spanning large time leaps in shorter time intervals. They provide us with means of performing simulation-enhanced thought experiments aimed at improving our intuition and understanding about the modeled phenomenon.

Our executable specification is based on the Abstract State Machine Language, introduced in Section 10.1. Section 10.2 provides an understanding of the structure and derivation of the AsmL model; Sections 10.2.1 through 10.2.5 present the gist of the AsmL model. The entire AsmL model can be found in Appendix C.

10.1 Abstract State Machine Language (AsmL)

To deploy ASMs in an industrial environment, we need an industrial-strength language; AsmL is one such language [54]. Abstract State Machine Language (AsmL) [74] is a high-level executable specification language based on the theory of Abstract State Machines. It is used for creating human-readable, machine-executable models of a system in a way that is minimal and complete with respect to any desired level of abstraction. The syntax and semantics of AsmL conform to the ASM modeling paradigm.
The language is being developed by Microsoft Research; the current version, AsmL 2, is embedded into Microsoft Word and Microsoft Visual Studio.NET.

Asml is object-oriented, strongly-typed, case-sensitive and uses indentation to denote block structure. AsmL has a rich type system containing type constructs for *sequences*, *maps*, *sets*, that support a number of high-level set-theoretic and sequence-theoretic built-in operations.

Consistent with the ASM theory, AsmL provides a parallel execution environment, whereby all statements within a scope are executed in parallel (i.e the order of execution does not matter); sequentiality is introduced by explicitly using the keyword *step*. An AsmL *program* is defined using a fixed *vocabulary* of symbols of our choosing. It has two components: the names of its *state variables*, denoted by the keyword *var*, and a fixed set of *operations* of an abstract state machine. *State* can be seen as a particular association of variable names to *values*. A *run* of the machine is a series of states connected by *state transitions*. Each state transition, or *step*, occurs when an *operation* is applied to an input state and produces an output state. The *program* consists of *statements*. A typical statement is the *conditional update* 'if condition then update'. Each update is in the form 'a := b' and indicates that variable name a will be associated with the value b in the output state. The program never alters the input state, instead each update statement adds to an *update set*. Pending updates are not visible in any program context, but when all program statements have been invoked, the pending updates are merged with a copy of the input state and returned as the output state. An *inconsistent update* error occurs if the update set contains conflicting information.

AsmL has the construct *forall* to support parallelism. Such constructs are required for high-level modeling, where it may be desirable to abstract from sequentiality or the order

of execution of statements. The current version of AsmL does not have runtime support for true concurrency; instead concurrent behavior is simulated by means of interleaving the steps of the agents. The AsmL construct *choose* construct can be used for making non-deterministic decisions, where it is not required to model algorithmic patterns behind the selection criteria.

The basic introduction provided here should serve the purpose for understanding the AsmL specifications that follow. For a more detailed understanding of the language, the reader is referred to [74] and [54].

10.2 Overview of the AsmL Model

In rendering an executable semantics to our DASM ground model, we advocate the use of AsmL. AsmL is a rich language, with advance language constructs, and close in semblance to the ASM modeling paradigm.
We use the AsmL specification primarily with the aim to explore and validate the requirements and design, and to test the conformance of the implementation with the specification. As a secondary outcome, the executable specification is used as a simulation tool, by building a graphical visualization on top of it. Our main goal is to establish a minimal, yet principal executable model to reveal the feasibility of achieving such a model through refinement. Through such a model we also show the importance of executable specifications in early design stages and prove how simulation and testing, by using such a model, can provide useful feedbacks to establish key system attributes.

It is worth mentioning that although the current executable model covers the ASM ground model at the given level of abstraction, it does not do so in its entirety. Some abstract functions (in the TSM and ADM) need to be refined further to derive true real-time and emergent behavior of the agents. These functions call for interfaces to sophisticated problem-solving and decision-making techniques, which are beyond the time scope of this thesis. There is also a need for the use of sophisticated programming and visualization techniques, such as event-driven programming, GIS etc, to supplement the operational and graphical results. It is anticipated that this work will be carried forward by the research group in near future, to give it a complete picture.

Refining the DASM ground model to the AsmL executable model requires considerations with respect to the translation aspects, refining abstract parts of the model, and achieving a useful method of visualization.
Intuitively, the AsmL encoding splits into five separate chunks, each of which deals with a basically different level of abstraction and distinct part of the ASM model: (1) the *Global Definitions* as explained in Section 10.2.1 are the major ASM data structures that are used throughout the model, these are primarily the data structures of the ASM Abstract Model (2) the *AsmL Abstract Model* as explained in Section 10.2.2 is the translation of the modules of the ASM Abstract Model (3) the *AsmL Refined Model* as explained in Section 10.2.3 is the translation of the ASM Refined Model (4) *Execution-Specific Additions* as explained in Section 10.2.4, and (5) *Visualization Specific Additions* as explained in Section 10.2.5.
In the following sections we illustrate the translation aspects that require heed in order to derive an AsmL model. While we try to maintain a congruent relation between the ASM and AsmL models, some changes are but inevitable.

10.2.1 Global Definitions

The global definitions entail defining the Entities used in the model, Environment (subjective and objective), the Person Agent Architecture, the Signals, the domain Time, and other miscellaneous and auxiliary data structures. In this section we illustrate some translation aspects by explicating the first three global definitions; the definitions for latter three can be found in Appendix C.1.

Linking Social Systems to DASM Models

The basic entities used in modeling the underlying sociological system through DASM models embodies defining some basic entities at the three levels. The ASM definitions as given in Chapter 5 translate to the AsmL data structures as shown in Figure 10.1.

It can be seen that all ASM domains for different entities are specified as of AsmL data type *Class*. *DASM AGENT* represented as *Class AGENT* in the AsmL model has a *Program* associated with it, which defines the behavior of this agent. All instantiations of this

```
//*************DASM Entities**********
public class AGENT
 virtual Program()

//*********MAS ENTITIES***************
type ENTITY = PASSIVE_OBJECT or ACTIVE_OBJECT or AUTONOMOUS_AGENT

public class PASSIVE_OBJECT
 // Attributes are represented by static functions

public class ACTIVE_OBJECT extends PASSIVE_OBJECT
 //Behavior is represented by dynamic functions

//This class should ideally extend both AGENT and ACTIVE_OBJECT.
//However, current AsmL version does not support multiple inheritance.
public class AUTONOMOUS_AGENT extends AGENT //extends ACTIVE_OBJECT
 // Motivations are represented as abstract/derived functions.
 // Memory is represented as functions.
 // Behavior is represented by the Program()
```

Figure 10.1: *AsmL Spec for Basic Entities.*

class *override* the *virtual* function *Program* to define agent-specific behavior. Clearly, Autonomous Agent *is a* kind of Agent and hence *extends* or *inherits* the class Agent.
The translation aspects discussed here with respect to Agents apply to other data structures having similar properties.

Environment

The Geographic Environment and Subjective Environment as defined in Chapter 6 translate to the AsmL model as described in this section.
The underlying graph called ENVIRONMENT_GRAPH in the ASM model is shown in Figure 10.2. The domains NODE and EDGE are represented as AsmL *class* and *structure* respectively. The domain ENVIRONMENT_GRAPH is defined as a *class* with *variables* that represent the set of nodes and edges contained in the graph. Various derived functions defined on the ENVIRONMENT_GRAPH are represented as *member functions* of the class.

The domain GEOGRAPHIC_ENV can then be defined as shown in Figure 10.3. Each ASM

```
//Class NODE representing NODE of the GRAPH
public class NODE

//Structure EDGE representing EDGE of the GRAPH
public structure EDGE
 edgeHead as NODE
 edgeTail as NODE

//Class GRAPH representing ENVIRONMENT_GRAPH
public class GRAPH
 var nodeSet as Set of NODE = {} //All the nodes of the GRAPH
 var edgeSet as Set of EDGE = {} //All the edged of the GRAPH

 //Returns the set of edges incident to the given NODE
 outIncidentEdges(node as NODE) as Set of EDGE
 //Returns whether the given NODES form an EDGE
 adjacent(u as NODE, v as NODE) as Boolean
 //Returns the EDGE formed by the two given NODES
 edge (u as NODE, v as NODE) as EDGE?
```

Figure 10.2: *AsmL Spec for GRAPH.*

domain representing the basic ATTRIBUTES (e.g GEO_STATIC_ATTRIBUTE) is translated to AsmL data type *enum*, where the *enum values* represent the actual factors (e.g traffic, distance). The AsmL data structure *type* is used to denote a union (by *or*) of different domain types. Finally, GEOGRAPHIC_ENV is represented as a *class* that *extends* the GRAPH. The *variables* of this class which are of type *map* represent the attribution schema.

The definition of SUBJECTIVE_ENV is similar to GEOGRAPHIC_ENV. The only difference being since it is agent-specific, it is defined inside the class agent as can be seen in the next section.

```
//***Domains for ATTRIBUTES***
enum DYNAMIC_NODE_ATTRIBUTE //1..10

enum STATIC_NODE_ATTRIBUTE //11..20
        coordinate = 11
        nodeName = 12

enum DYNAMIC_EDGE_ATTRIBUTE //21..30
        traffic = 21
        roadCondition = 22

enum STATIC_EDGE_ATTRIBUTE //31..40
        distance = 31
        roadType = 32
        edgeName = 33

type GEO_EDGE_ATTRIBUTE = DYNAMIC_EDGE_ATTRIBUTE or STATIC_EDGE_ATTRIBUTE
type GEO_NODE_ATTRIBUTE = DYNAMIC_NODE_ATTRIBUTE or STATIC_NODE_ATTRIBUTE
type GEO_ATTRIBUTE = GEO_EDGE_ATTRIBUTE or GEO_NODE_ATTRIBUTE

//***Class representing the GEOGRAPHIC ENVIRONMENT***
public class GEOGRAPHIC_ENV extends GRAPH
  var geoStaticNodeAttr as Map of NODE to
        (Map of STATIC_NODE_ATTRIBUTE to VALUE) = {->}
  var geoStaticEdgeAttr as Map of EDGE to
        (Map of STATIC_EDGE_ATTRIBUTE to VALUE) = {->}
  var geoDynamicEdgeAttr as Map of EDGE to
        (Map of DYNAMIC_EDGE_ATTRIBUTE to VALUE) ={->}
  var geoDynamicNodeAttr as Map of NODE to
        (Map of DYNAMIC_NODE_ATTRIBUTE to VALUE) ={->}
```

Figure 10.3: *AsmL Spec for GEOGRAPHIC_ENV.*

Person Architecture

The Person Agent Architecture as described in Section 7.1.2 of Chapter 7 using ASM formalism, translates to the AsmL spec as shown in Figure 10.4. The domain MODULE is represented as a *class* that *extends* DASM Agent AGENT since it has associated behavior. The different kinds of modules (e.g TSM, SEM) are also represented as *classes* that *extend* MODULE.

The autonomously acting entity PERSON is implemented as a *class* that *inherits* AUTONOMOUS_AGENT and has the modules associated with it declared as *variables* of the class. The subjective environments is composed of different attributions that are represented

as variables of type *map*. The virtual *Program* of the DASM AGENT is *overridden* by the PERSON class — the *step until fixpoint* construct is used to loop through the following statements to emulate the fact that the PERSON is continuously working.

```
//Each Module has its corresponding parent agent.
public class MODULE extends AGENT
 var parentAgent as PERSON = null

//Different Modules.
public class TARGET_SELECTION_MODULE extends MODULE //TSM
public class SPACE_MODULE extends MODULE            //SEM
public class DECISION_MODULE extends MODULE         //ADM

public class PERSON extends AUTONOMOUS_AGENT
 var spaceModule as SPACE_MODULE = null //SEM
 var targetModule as TARGET_SELECTION_MODULE = null //TSM
 var decisionModule as DECISION_MODULE = null //ADM

 //***Subjective Environmnet***
 //NOTE: SubjEnv can be declared as shared in person agent OR
 //as a global var to capture ASM global state.
 var subjEdgeAttr as Map of EDGE to (Map of SUBJ_EDGE_ATTRIBUTE to VALUE) = {->}
 var subjNodeAttr as Map of NODE to (Map of SUBJ_NODE_ATTRIBUTE to VALUE) = {->}

 // ***Kept in Working Memory ***
 var currentNode as NODE = null
 var currentEdge as EDGE = null

 //***Program***
 override Program()
  step until fixpoint
   me.decisionModule.Program()
   me.spaceModule.Program()
   me.targetModule.Program()
```

Figure 10.4: *AsmL Spec for PERSON AGENT.*

10.2.2 AsmL Abstract Model

The AsmL Abstract Model is composed of the three modules viz. Space Evolution Module (SEM), Target Selection Module (TSM) and Agent Decision Module (ADM). In this section we illustrate the translation aspects w.r.t the SEM only. The same principles apply for TSM and ADM as can be seen in Appendix C.2.

First, the definitions of the SEM are described in Figure 10.5. The different modes associated with the SEM are grouped using the AsmL data type *enum*; the *enum values* represent the individual modes. SEM is implemented as a *class* whose member data types (*var*) represent the functions associated with the SEM. Derived functions are represented as *member functions* of this class.

```
enum Mode
  idle
  pathPlanning
  roadSelection
  localRePlanning
  running
  pathCompleted

public class SPACE_MODULE extends MODULE
 //********** KEPT IN WORKING MEMORY ***************
  var mode as Mode = idle // Mode of SEM

  var destNode as NODE = null  // Current destination.
  var sourceNode as NODE = null  // The source of the path.
  var suggestedPath as PATH = []  // Path suggested for traversal
  var suggestedEdge as EDGE = null  // Edge suggested for traversal
  var takenPath as PATH = []  // The final path person actually traverses.

// ******Derived Functions******
  currentNode() as NODE  //current node as stored in Person
  currentEdge() as EDGE  //current edge as stored in Person
  destNodeReached() as Boolean  // Calculates if the current destination reached
  signalFromADM() as Boolean  //Checks if there is any signal from ADM.
  currentEdgeTraversed() as Boolean  // Determines if current edge has been traversed.
```

Figure 10.5: *AsmL Spec for SEM Definitions.*

The SEM Program *overrides* the *virtual* function *Program* of AGENT (Figure 10.6). There are a set of parallel $if s$ corresponding to each mode of the SEM. At any point of time only one of the modes is active; SEM is initialized to be in idle mode. At the end of each mode, the next immediate mode is set as the active mode. The abstract rules used by the SEM are written in block letters. At the next level of refinement, these abstract functions are detailed. The SEM goes through a number of intermediate refinement steps.

```
public class SPACE_MODULE extends MODULE
 //************** SPACE MODULE PROGRAM ***************************
 override Program()

  if me.mode = idle then
    let x = SIGNAL.OnSignal(newDest,me)
    if x <> null then
      let s = x as NEW_DEST
      INITIALIZE(parentAgent.currentNode, s.newDest)
      me.mode := pathPlanning
    else
      skip

  if me.mode = pathPlanning then
    GET_PATH()
    mode := roadSelection

  if me.mode = roadSelection then
    if destNodeReached() = true then
      me.mode := pathCompleted
    else
      if signalFromADM() then
        HANDLE_ADM_SIGNALS()
        mode := pathPlanning
      else
        GET_SUGGESTED_EDGE()
        mode := localRePlanning

  if me.mode = localRePlanning then
    if acceptableEdge(suggestedEdge) then
      me.parentAgent.currentEdge := suggestedEdge
      me.mode := running
    else
      me.mode := pathPlanning
    RECORD_SELECTED_EDGE()

  if me.mode = running then
    if me.currentEdgeTraversed() then
      UPDATE_EDGE_PERCEPTION(parentAgent.currentEdge)
          FINALIZE_EDGE_TRAVERSAL()
```

Figure 10.6: *AsmL Spec for SEM Program.*

10.2.3 AsmL Refined Model

The AsmL Refined Model is the AsmL version of the ASM Refined Model, and is composed of the definitions of the Abstract CBR and Concrete CBR; and the Path Explorer submachine. The translation aspects related to these two components are the same as described above, and so we do not further elaborate on them. Interested reader is referred to Appendix C.3.

10.2.4 Execution Specific Additions

In order to make the model run and produce output, we need to carry out further refinement steps. While there are a number of complex execution specific additions (Appendix C.4), we concentrate here on the function Main(). The function *Main* is the *entry point* of the AsmL model and is shown in Figure 10.7.

```
Main()
  step
        InitGraphXML() // To initialize Geographic Environment from XML File
  step
    //Initializing PERSON Properties
    let person1 =
    InitPersonAuto("0","0", 1.0, 1.0, 0.0,
     {frequency ->1.0,tripImportance->0.0,reinforcement ->0.0,intensity->0.0,
      distance-> 1.0, roadType ->0.0,edgeName->0.0, angle ->0.0,
      numberOfStops ->0.0, traffic ->0.50 },
      true, node("0"), node("19"), node("24"))
        //create other person agents...
  step
    add person1 to personSet
        //add other agents...

  step until exit = true
    forall p in personSet //Run the Program for each Person
      p.Program()
```

Figure 10.7: *AsmL Spec for Main().*

First, it initializes the geographic environment from an XML repository using the function InitGraphXML. Next, it creates a number of person agents using the function InitPerson-Auto and adds all these agents to *personSet*. Then for each person agent in the person set, it uses the AsmL construct *forall* to run the program of all agents concurrently. Finally

the time is increased to the next time step and the program keeps running until an exit condition is met.
Both the functions, InitGraphXML and InitPersonAuto are complex functions that use a number of other subordinate functions. InitGraphXML reads an XML file into an interim data structure and then initializes the geographic environment related data structures from this interim data structure. InitPersonAuto calls a number of other functions to initialize person's SEM, TSM, ADM, subjective environment etc.

Another important aspect that needs to be addressed in the stepwise refinement of the model, is of refining the abstract functions acting as 'oracles'. These oracles are non-computable functions and clearly in order to have a meaningful executable model, these need to be translated into computable functions. Abstract rules are refined either by introducing non-determinism using the AsmL construct *choose* or by assigning pseudo deterministic behavior to them.

10.2.5 Visualization

For effective and meaningful projection of results as produced by the AsmL executable model, a user-friendly graphically enhanced visualization is needed. Such a visualization is also a tool for user-controlled simulation and testing.
For the purpose of this thesis, we use a modest visualization implemented in C++ and OpenGL. This visualization has been developed by Steven Bergner, a PhD candidate in the SFU Gruvi Lab.
The AsmL model communicates with the Visualization using a number of commands. Both the AsmL model and the visualization utilize an XML based data structure of the urban area. The communication commands and the XML graph are detailed in Appendix C.5.
Some snapshots of the visualization are provided here.

Figure 10.8 shows the primary view of the visualization — aggregate activity space and crime occurrence space for all agents and all crime types. The geographic environment is composed of a small Vancouver Downtown area containing 32 nodes and 40 edges. The connecting roads (edges) are in bright blue; major roads appear thicker than minor roads for distinction. There are six person agents (0-5) initially located on this map at different node locations. Different colors represent different agents (e.g person 0 is blue in color);

filled small boxes represent current location and unfilled boxes represent current destination of a given agent. The *activity spaces* of the agents are shown in yellow along the edges. The intensity of color represents the aggregate spaces of different agents, and the spread represents the strength (frequency) of the activity space. The colored boxes located on the nodes show the aggregate *crime occurrence space* of the different agents; the spread of the box represents the probability of crime. Red colored boxes represents car theft probability, green represents shop lift probability and black represents robbery probability. At bottom left corner, one can see the current time of the simulation; this slot is also used for showing some miscellaneous information during simulation runs.

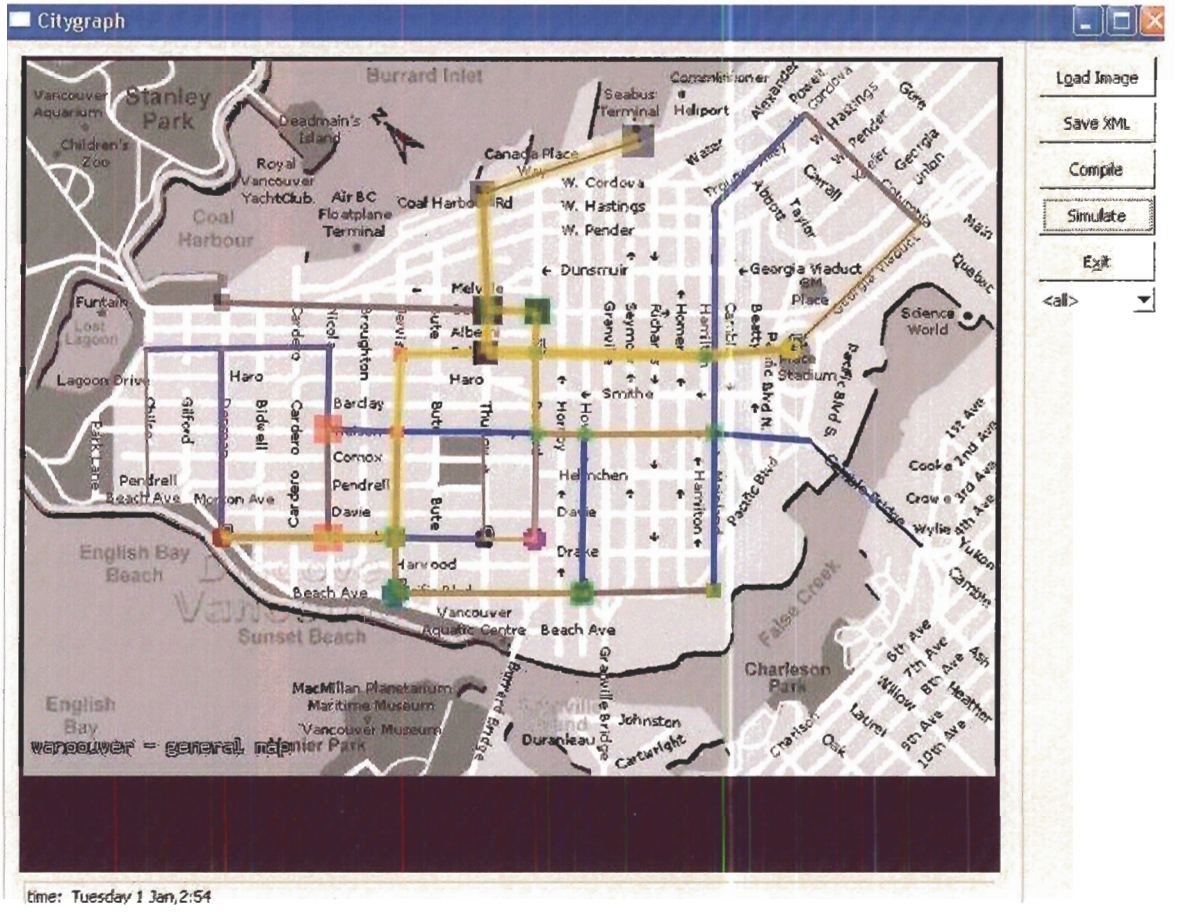

Figure 10.8: *An Aggregate View of the Visualization.*

The visualization can be also be viewed for a specific crime type for all agents. Figure 10.9, Figure 10.10 and Figure 10.11 show the car theft, shop lift and robbery probabilities respectively. The shaded area in gray in each of these snapshots represents the predefined *opportunity spaces*. As can be deduced, red boxes are coincident with the car theft opportunities and represent the probability of car theft; green boxes represent shop lift probability and black boxes represent robbery probability.

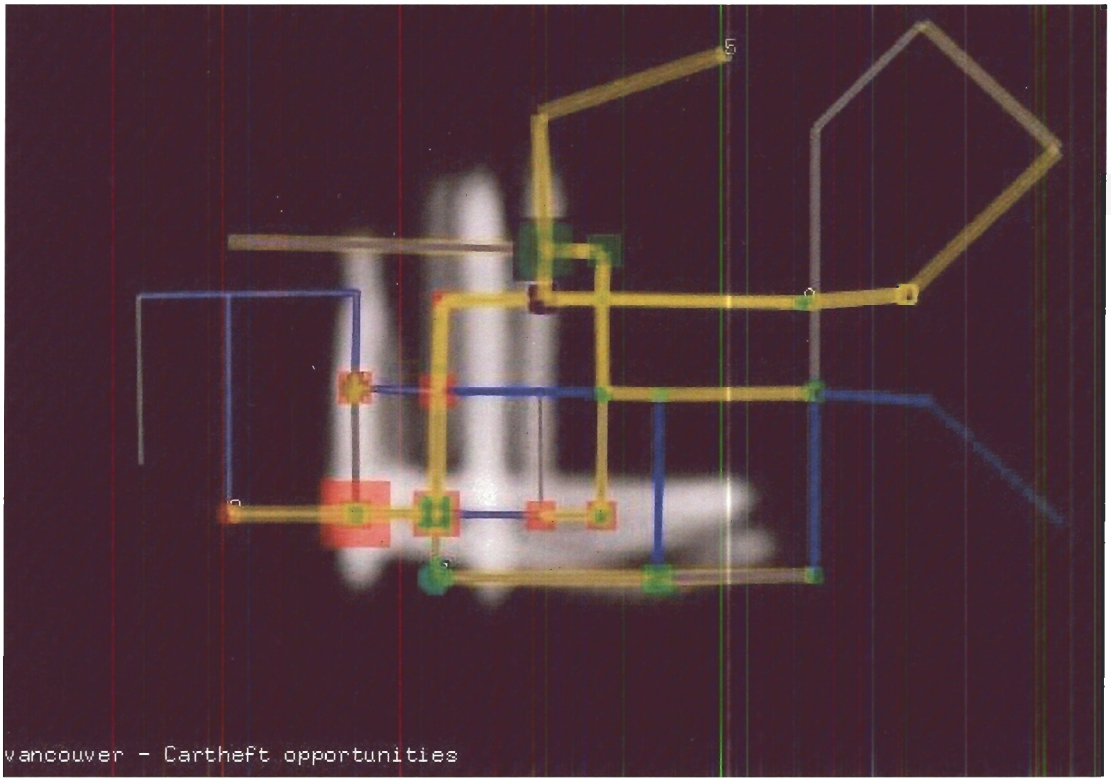

Figure 10.9: *Car Theft Opportunities and Probability.*

Lastly, one can also view the visualization for a specific agent for aggregate crimes or a specific crime. This however is not shown here.

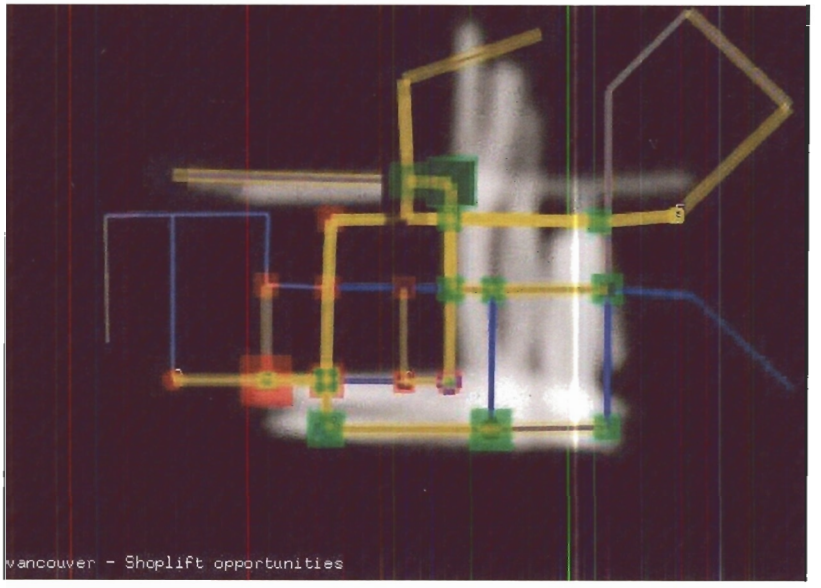

Figure 10.10: *Shop Lift Opportunities and Probability.*

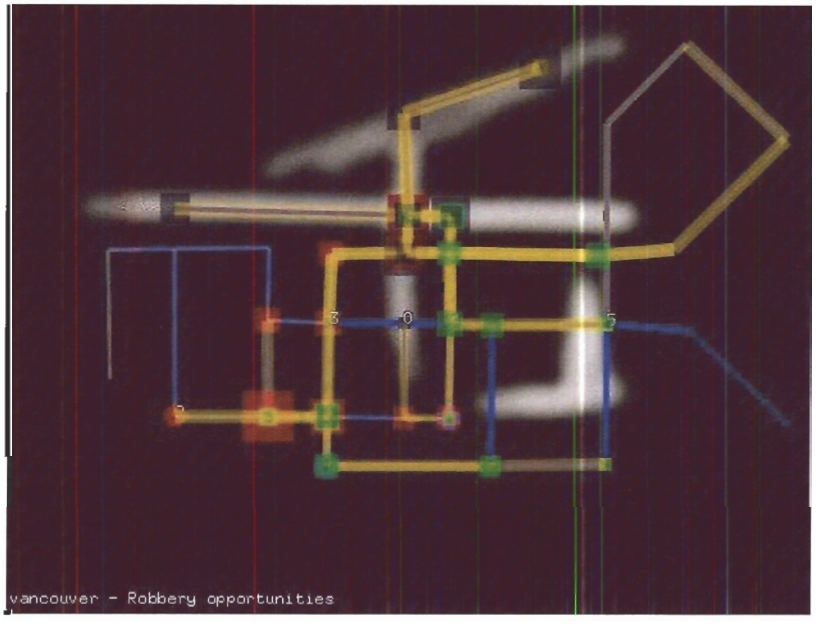

Figure 10.11: *Robbery Opportunities and Probability.*

Chapter 11

Validation of the Model

This chapter discusses the need for validation of the model, particular challenges and hurdles met, possible evaluation techniques and finally some preliminary experimental results.

11.1 Overview

Computational and mathematical modeling of real-world phenomena is meaningful only if one can establish the validity of a model and assess the quality of the results it produces. For descriptive modeling, the problem of providing evidence for the validity of a model has a different meaning than in prescriptive modeling where quantitative measures of reliability of predictions apply [18].
However there are even greater problems involved in validating models of human societies. Even though the model has been experimentally validated, just because the model is only a model, it will always be possible to dispute any parallel claims between its behavior and of the target [52].

An obvious open-ended question is then how can one validate such a model and prove its soundness from both a criminological and a computational perspective. Apparently, there is no simple way to do this, but there is also no better alternative other than modeling. One can combine *qualitative* and *quantitative* evaluation techniques.

Quantitatively, one can evaluate the model in two different ways. One is to compare results

of simulation runs to results produced by other prediction models using statistical methods as proposed in [56] and attempt to reduce the observed error and deviance. The other way is to use real-world data from crime databases as a basis for comparison. In this case, the goal is to reproduce results matching those derived from real-world data of comparable scenarios. For instance, in software cost estimation, a widely accepted rule of thumb suggests that a model is acceptable if 75% of the predicted values fall within 25% of their actual values [43]. Both approaches provide feedback mechanisms for calibrating the various model parameters. The latter approach is being focused upon as another research project in the ST Lab [1].

Qualitatively, typical purposes of such experimental research, among others, include [87]: a) use of a model for generating and testing hypothesis through controlled experiments, b) use of a model for predicting the effects of change in the system under study. In this context, the role of modeling and simulation is descriptive rather than prescriptive and, for instance, focuses on aspects such as: a) identification of behavior characteristics from response patterns generated by simulations, and b) identification of the system boundary and the factors that influence the behavior of interest. Clearly, the key issue is not to obtain a quantitative simulation response as the main result but to inspect the underlying trace that generated the response and suggest changes on that basis.
An obvious way of validating the model is by simulating scenarios and comparing the results with what the theories predict. However, there is no guarantee that the theories make correct predictions under all circumstances. A possibility is the use of compositional techniques with the idea to divide the validation task by applying different approaches specific to each of the model components (e.g., cross-validation can be used for validating neural networks). The problem is then, how to validate coherence and consistence of the integration of components.

Another qualitative aspect includes addressing the consistency of both the symbolic representation of the model as well as its definition with regard to the underlying theories of crime. We exploit the concept of *ASM ground models* [6] — an abstract, complete, precise and yet understandable mathematical model – and by carefully analyzing and eliciting requirements is the best one can do in the overall attempt toward making the semantic model as sound and complete as possible.

[1] Another interesting possibility is to use the crime databases for past years, for e.g. 2000-2004, to build the model and subsequently use the model to predict 2005-2006 crimes.

Some authors have developed qualitative criteria in evaluating agent-based systems of human societies.

Gilbert and Doran [52] highlight certain aspects that must have an impact upon the modeling and simulation of human societies — *environment, complexity, distribution, cognition, communication.*

She and Stu [85] develop a criteria to measure the performance of agent-based modeling techniques. The criteria is divided into software engineering characteristics (*preciseness, accessibility, expressiveness, modularity, complexity, executability, refinability, analyzability, openness*) and agent-based characteristics (*autonomy, complexity, adaptability, concurrency, distribution, communication*).

The authors of [34] contend that an agent formalism should have the following characteristics: *precise* and *unambiguous* language; move in a principled way from *specifications to implementations*; deal with *multiplicity* of agents; address the needs of *practical applications* of agents by being capable of expressing some or all of the aspects of agency such as perception, action, belief, knowledge,goals, motivation, intention, desire, emotion, etc; help identify properties of agent systems against which implementations can be measured and assessed; measure, evaluate, classify, and study implementations.

To this end, the ASM formalism and abstraction principles in combination with the underlying methodological framework of ground models and refinement techniques provide a universal formal basis for semantic modeling of multi-agent human societies at arbitrary levels of abstraction in a coherent and consistent framework. The theory of ASM modeling paradigm successfully captures most if not all of the aforementioned key issues.

11.2 Experimental Validation

Based on the above rhetoric, one can conclude there is apparently no simple way to validate such a model, but there is also no better alternative other than modeling. In this section on experimental validation, we make an attempt to report on some observations made while simulating typical scenarios.

It should be noted that the model is still abstract and non-deterministic measures are used as oracles that produce results which otherwise require sophisticated techniques. This stage of infancy and abstraction inhibits the capability of the model to produce 'true' emergent

results, we merely experiment with artificial and 'dummy' data. Further we use only a prototype of the simulation implemented in a specification language that uses high-level data structures, and thus the luxury of sophisticated data structures implemented in programming languages for time efficiency cannot be enjoyed.
Therefore, this attempt on reporting some observations should not be considered as a hard and fast statement of the experimental validation and verification of the model. This aspect deserves a more thorough investigation, and is beyond the time scope of this thesis, and is subject of our future work. These observation however do provide an insight on the soundness of our model.

The goal of the executable model is to simulate the movement patterns of hypothetical offenders in a given urban environment that evolves their activity spaces over a period of time. Based on the activity spaces, the model carves out the crime occurrence spaces of the criminals for different crime types (burglary, shop lift, car theft).
We start with a prototype of Vancouver Downtown area. Currently, the graph consists of 32 nodes (vertices) and 40 edges. Although minimal, it represents a typical urban area; the graph size can potentially be extended to any number of nodes and edges.
These edges area all bidirectional. With each edge is associated information about its geographic attributes, both static and dynamic — distance, road type (major, minor), edge name, traffic (low, medium, high, blocked). With each node is stored information about its geographic attributes — coordinates, node name. This forms the universal Geographic Environment (GeoEnv).
With each agent is associated its subjective environment. We initialize the agent-specific subjective environment as this. For the perception attributes, which are perceived or interpreted values of geographic attributes, we initialize them to be the same as geographic attributes[2]. For the activity attributes — frequency, reinforcement (positive, negative, neutral), trip importance(obligatory, required, not required) — the values are initialized to default values as 0, neutral, not required respectively. For the awareness attributes — intensity — the value is initialized to default as 0.
As and when an agent moves around in the given environment, its subjective environment starts building up, i.e we update the values of perception, activity and awareness attributes.

[2]Ideally, perception attributes should represent interpreted values of geographic attributes which are different from the actual values. However, since we use abstract functions this can be achieved easily in the later refinements.

For the perception attributes, the old values in memory are updated to be the current geographic values. For frequency, the count is increased by 1. For reinforcement, trip importance, intensity, the updated values are an average of old values in memory and the current values; this however is achieved by abstract functions. The simulation at its preliminary stage carves the activity space of the different constituent agents. Each agent is endowed with its activity schedule, although ideally this schedule should be derived from a person's demographic factors. The ADM_SEM_MONITOR provides the next destination to be traveled to based on the schedule. The abstract data structures for schedule allows for probabilistic destinations to be associated with any given time, i.e an agent probabilistically chooses the next destination; however, in this version, we merely associate a probability 1 with each destination. The SEM then moves the agent to the proposed destination. Hence, the SEM and ADM_SEM_MONITOR collectively model the spatial and temporal aspects of movement of person agents in the given urban landscape.

Using a predefined opportunity space and an agent's activity space[3], the simulation produces crime occurrence space of an agent. The abstract ADM_TSM_MONITOR decides whether the agent is criminally disposed based on its abstract motivations. If so, the SEM module is called. At this point, the SEM non-deterministically selects those targets that are on locations which are above a certain threshold of the activity space. The crime occurrence space is a function of the criminal opportunities(car theft, robbery, burglary) located on the nodes of the graph, and the criminal skills of the agent for that particular crime. Hence, the TSM and ADM_TSM_MONITOR collectively model the spatial and temporal aspects of target selection of person agents in the given urban landscape.

All simulation runs were carried out on an IBM PC with an Intel Pentium 4 processor of 2.40 GHz, and 1.0 GB of RAM.

[3]Ideally, awareness space should be used instead of activity space. However, since activity space is a proper subset of the awareness space and simpler to compute, we restrict to this case.

11.2.1 Space Evolution Module (SEM)

Time Performance of the Shortest Path Algorithm

In finding a shortest path for an agent from a given source to a given destination, we use the 'shortest path algorithm' as developed in Chapter 9, which combines global planning with local negotiation; the algorithm can also be seen as a combination of A* and Dijkstra. In our shortest path algorithm, the dominant cost is of the Dijkstra's algorithm to calculate the global path preference of the agents; this entails computing an all source shortest path for an agent. The complexity of our algorithm is thus $O(n^3)$, where n is the number of vertices in the graph. The navigation algorithm as used in the SEM performs real-time local re-planning at each intervening node if the suggested edge is found to be non-acceptable or the influence factor weights have changed, in which case the 'shortest path algorithm' is executed again. In the worst case, all of n nodes might be on the suggested path and for each intervening node, one might have to do local re-planning. The time complexity thus increases to $O(n^3 * n)$. Further, since each agent has different influence factor weights that determine the cost of the paths, in the overall simulation where all agents are running concurrently, the time delay may increase to $O(n^3 * n * a)$, where a is the number of agents in the simulation. $O(n^3 * n * a)$ clearly is not a negligible amount of time delay. Hence, the performance of the 'shortest path algorithm' is a bottleneck factor in the performance of the Navigation algorithm.

We ran our simulation with a graph size of 32 nodes and 40 edges, and gradually increased it to 63 nodes and 74 edges. Tabulated below (Table 11.1) are the time responses of the shortest path algorithm. The reported time for each graph size is an average of ten runs to different destinations for different agents. The cost of each edge (weighted sum of factors) is precomputed and hence the time delay for this calculation is negligible.

Graph Size	Response Time
32 Nodes, 40 Edges	3.5 sec
42 Nodes, 50 Edges	14 sec
52 Nodes, 62 Edges	47 sec
62 Nodes, 74 Edges	2.15 mts

Table 11.1: *Response Time of the Shortest Path Algorithm w.r.t Graph Size.*

Thus with increasing graph size, the computation time increases drastically. This small

experimentation has lead us to consider optimizations techniques to increase the time efficiency. Two kinds of optimizations can be performed : (1) using special data structures for computation of the Dijkstra's algorithm and (2) using a hierarchical graph structure.
Various implementations of the Dijkstra use data structures such as buckets, heap structures, hash tables in increasing the speed efficiency [26], [75]. Such advanced data structures however require the luxury of a programming language.
By using a hierarchical graph model, the underlying structure of the complex topographical map can be exploited, that while path calculation may restrict search space to a sub-set of the graph, and thus result in search time and speed benefits. [88], [63] use such a structure and develop algorithms that significantly reduce the search space and time.
Implementing theses optimization techniques are part of our future work.

Time Performance of the Navigation Algorithm

The SEM implements a hierarchical navigation algorithm, that integrates CBR and a shortest path algorithm, based on Dijkstra. This is detailed in Section 7.2 of Chapter 7. We make a few observations on the time performance of the algorithm with respect to the different cases it represents. A graph of 32 nodes and 40 edges is used.

In the first case, if there is a path stored in the persons' memory (CBR) for a given source to destination, it is returned instantly with negligible time delay. This as opposed to the shortest path algorithm saves 3-4 seconds.

In the second case, if there is not an exact match, a path close enough to the destination is returned. This again takes negligible computation time. In computing the remainder of the path however, the shortest path algorithm is called for the entire graph. This nuance can easily be resolved by considering hierarchical graphs (as mentioned above), whereby the search area can be pruned and only a very small subset of the graph containing both the ultimate destination and the interim destination is considered. Since we do not have a hierarchical graph, we emulate this time saving by using the pre-stored global path preference stored as $globalPathPref$. This case takes more time than the first case, however it takes much less time than the third case where no exact or partial match exists in the CBR.

In the third case, if there is no exact or even partial match in the CBR, the shortest path algorithm is called that runs on the entire graph, or in case of hierarchical graphs on a

subset of the graph. This takes the most time, between 3 - 4 seconds for a given path.

Thus, the ordering of the three levels of the hierarchy also represents increasing time delay as shown in Table 11.2.

Case	Response Time
Pure CBR (exact match)	$\tilde{0}$ sec
Mixed Case (partial match)	0 - 4 sec
Shortest Path Algorithm	3 - 4 sec

Table 11.2: *Response Time Variations for Three Cases of the Navigation Algorithm.*

It is observed experimentally which fulfills the natural expectation that initially when an agent is 'new' to its surrounding, it takes a long time in trying to figure out which paths to take. There is a learning curve and after a while the movement patterns become relatively fixed and self reinforcing; the agent merely recalls from memory all path planning decisions.

In graphs of very large sizes, for e.g our experimental graph of 62 nodes which took nearly 2.5 minutes in computing a shortest path, such a time gain with a CBR technique can be a bottleneck factor. Although time efficiency is not of utmost importance in our model, in typical real-time transportation systems or in time critical applications such a time benefit can be highly desirable.

Characteristic Features of the Navigation Algorithm

All observations reported henceforth are performed on a subgraph of Vancouver Downtown area composed of 32 nodes and 40 edges as shown in Figure 11.1.

The navigation algorithm successfully demonstrates the hierarchical decision-making process — calling the CBR for an exact match (pure CBR), if not calling the CBR for a partial match and the shortest path algorithm for the remaining path (mixed CBR), and lastly calling the shortest path algorithm (Explorer). This is evident in Table 11.3 which is shown for a given agent (Agent 1) that is CBR dominant.

The algorithm uses a normalized weighted sum of a number of influence factors (distance, road type etc) in calculating the overall preference of an edge and subsequently the path. The different factors play a composite role in determining the overall edge preference where the influence of each factor is controlled by the influence factor weight (value between 0-1).

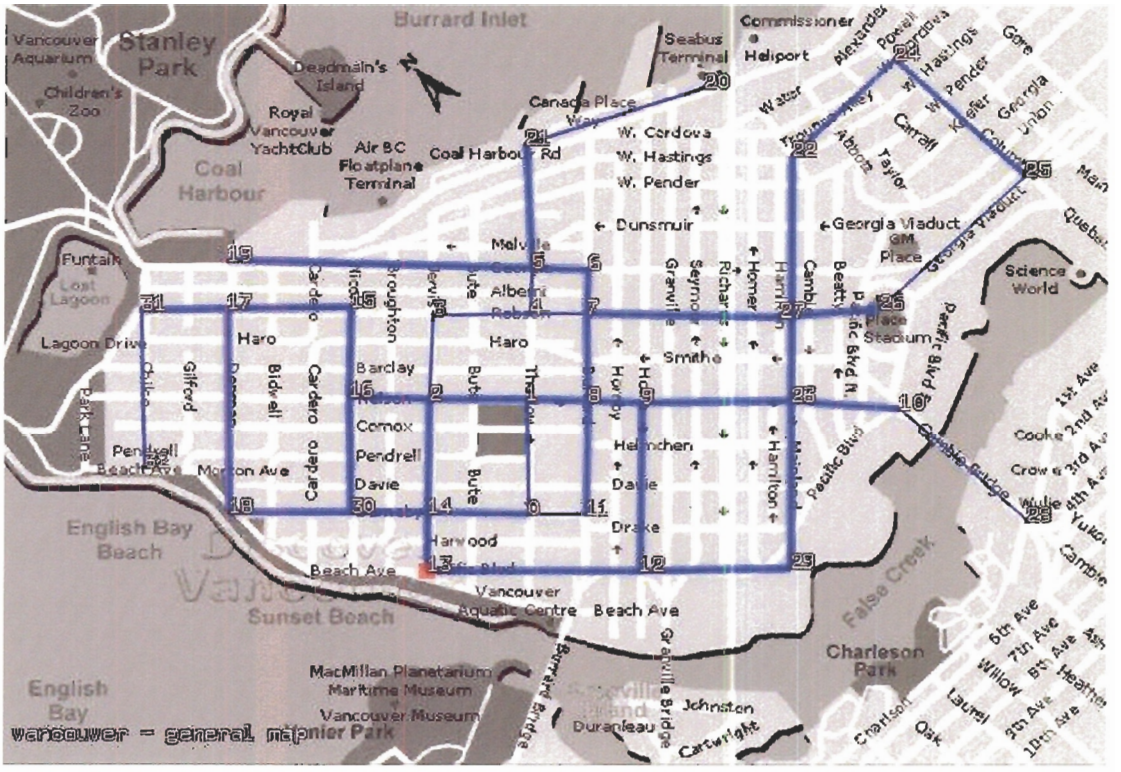

Figure 11.1: *Experimental Graph of Vancouver Downtown*

Run	Source	Dest.	Case Called	Path Returned
1	0	24	Explorer	0-11-8-9-23-22-24
2	24	19	Explorer	24-25-26-27-7-6-5-19
3	19	24	Pure CBR	19-5-6-7-27-26-25-24
4	24	19	Pure CBR	24-25-26-27-7-6-5-19
5	19	0	Explorer	19-5-6-7-8-11-0
6	0	10	Mixed CBR	[0-11-8] + [8,1]

Table 11.3: *Hierarchical Cases of the Navigation Algorithm.*

A very interesting observation that came to light while experimenting with the model was the fact that although the individual factor values were normalized to fall within a certain range, it was not taken care of that the values be also normalized with respect to the distance of the edge. For e.g let's take two edges A-B and C-D that are of same length and have low traffic on them, A-B does not have any intervening nodes, while C-D has two intervening nodes. Since C-D has two intervening nodes, the Dijkstra algorithm in computing the cost of this edge counts the traffic of this edge three times which might make the overall value of traffic high, whereas it should still be low. To avoid this, we also normalize the values of factors with respect to the length of the edge.

As mentioned before, the influence of each factor in the overall path preference is determined by the influence factor weight whose value is between 0-1. These values are different for different agents and typically depend on their demographic factors and personal liking. However, for general scenarios we use a weight value of 1 and 0 to indicate the fact that this factor plays or does not play a role in the overall preference, respectively. For e.g if a value 0 is assigned to the influence factor weight for road type, this implies the person has the same preference for minor roads as for major roads; if this value is changed to 1, this implies the person prefers major roads to minor roads.
We produced numerous runs of the simulation by altering the weights of the different influence factors and observed the path taken. Since there are a number of factors and all the factors together play an integrated role in the overall preference, it is not easy to determine whether the computed path is the actual desired path. The best one can do is analyze the influence of factors one at a time. We do so by setting all other factors to 0 and setting the influence of factor under study to 1; the path computed is recorded and the weight of this factor is then reverted back to 0 [4]. Clearly if alternate paths exist, the paths taken in the two cases should be different. Tabulated below (Table 11.4) are some observed results.

Thus, it is evident that the shortest path algorithm responds correctly to changes in influence factor weights. Since single factors produce correct results, it can be deduced that composite factors, when the weights of multiple factors is non-zero, also produces correct paths.

A point to be noted is on behavioral reinforcement that the algorithm successfully emulates.

[4]While the weights of all other factors are set to 0, the weight of Distance is still kept 1, to produce intuitive results

Factor Under Observation	Weight	Source	Dest.	Path Taken
Road Type	1	11	2	11-0-1-2
	0	11	2	11-0-1-2 11-8-1-2 11-0-14-2
# of Intervening Stops	1	29	7	29-23-27-7
	0	29	7	29-23-27-7 29-12-9-8-7
Distance	1	1	2	1-2
	0	1	2	1-0-14-2 1-8-7-4-3-2 ...many more

Table 11.4: *Influence of Factor Weights in the Shortest Path Path.*

If a path was taken once from a given source S to destination D, and if the agent encounters the problem of traveling from S to D (or a subset of this path) again, the invocation of the CBR returns this path (or sub-path). The same case holds for the mixed case. Exhaustive number of runs of the simulation demonstrate this faithfully time and again.

The navigation algorithm also successfully models non-determinism. Given that there are two paths from source S to destination D with the same value of preference, different runs of the simulation will return different paths and not just the same path over and over again. This holds for the path returned by the CBR as well. Table 11.5 demonstrates this fact.

Source	Dest.	Run No.	Path Taken
29	9	1	29-23-9
		3	29-12-9
9	13	1	9-12-13
		2	9-2-13
19	27	1	19-5-6-7-27
		5	19-5-4-7-27

Table 11.5: *Non-Determinism in the Navigation Algorithm.*

The algorithm also responds well to changes in the underlying environment. Since our visualization does not allow us to dynamically change the environment during a run, we checked the response to such changes by simulating different scenarios dis-jointly.

With time, the activity space of the agent grows. This is evident from the fact that initially the case-base associated with each agent is empty. It grows steadily with time, new paths

keep getting added, or the strength of added paths keeps increasing. Those agents that are more active have a larger activity space whereas those that are passive have a much smaller activity space. The size of activity space is the size of the SEM case-base, which is reported as the sum of distinct paths (cases) multiplied by their frequency. Refer to Table 11.6 for these results[5].

End of Day	Agent 1 (Active)	Agent 2 (Medium)	Agent 2 (Passive)
Monday	325	119	115
Tuesday	649	227	217
Wednesday	973	335	319
Thursday	1297	443	421
Friday	1621	551	523
Saturday	1849	667	559
Sunday	2107	759	625

Table 11.6: *Growth of Activity Space as Given by Size of Case Base.*

11.2.2 Target Selection Module (TSM)

The current TSM is at a very high level of abstraction. It checks to see if the target is a good target — if the location of the target is on an edge whose frequency (of activity space) is above a predefined threshold, it is then returned as a safe target or else a risky target. The simulation should thus produce this causal effect, whereby the working of the TSM depends on the activity space as created by the SEM. This causal effect does indeed happen.
With each agent is associated its 'crime likes', which is a value between 0 - 100 for the different crimes (car theft, shop lift, robbery). With each node (of the edge) is associated the targets located on that node called 'criminal opportunity'; which is a value between 0 - 100 for the different crimes. If a target is found to be suitable (i.e located on an edge above a certain frequency threshold) the probability of victimization is computed as a function of 'crime likes', 'criminal opportunity' and 'frequency'. We then decide whether the victimization of targets was successful or not based on the calculated crime probability and by using non-determinism; if it was successful the value of 'crime likes' is increased for that agent for that crime, or else decreased. This emulates the fact that agents reinforce

[5]It should be noted however, that with each path taken we store the possible combinations of this path as separate paths in the case-base, so indirectly the length of the path affects the size of the case-base.

their decisions positively or negatively depending on the outcome of their behavior.
Table 11.7 shows an output snippet for an agent that has initial 'crime likes' [cartheft = 100, shoplift = 10, robbery = 70] ; and activity edge threshold of 2 (i.e only considers targets located on edges that he has visited at least twice as good targets). We show some scenarios for edge 2-3 that has criminal opportunities on node 2 as [Cartheft = 38, Shoplift = 0,Robbery = 0] and 3 as [Cartheft = 38, Shoplift = 0, Robbery = 0]. Only car theft scenarios are shown here.

Run	Edge Activity Strength	Crime Likes	Crime Probability	Successful
1	0	38	n/a	n/a
2	1	38	n/a	n/a
3	2	38	38	Yes
4	3	40	45	Yes
5	4	42	55	Yes
6	5	44	60	No
7	6	40	50	Yes

Table 11.7: *Target Selection.*

PART V

PUTTING IT ALL TOGETHER

Chapter 12

Conclusion, Contribution and Challenges

In this thesis we have proposed a novel approach to modeling and simulation of crime patterns and theories in crime analysis and prevention — a key aspect in Computational Criminology [17]. This is achieved by combining the abstract state machine (ASM) paradigm for mathematical modeling of discrete dynamic systems, with multi-agent systems, and other problem-solving techniques.
The virtuosity of this work is extolled in its pioneering nature [19] [20] [18] [21]. It is the forerunner in rendering the theoretical field of Computational Criminology [17], [16] a pragmatic and a tangible base, sound both from a computational and a criminological perspective. To the author's best knowledge, there has been no former published research in Computational Criminology, of the magnitude presented in this work.

We exemplify our approach by modeling and simulating spatial and temporal aspects of crime in urban environments. Emphasizing the need for a well-defined and robust mathematical framework, we devise a distributed abstract state machine model as a formal basis for the development of simulation models. Although unconventional, the application of the ASM formalism and abstraction principles to social systems turns out to be a promising approach; it nicely combines with the established view of multi-agent modeling of social systems and provides a precise semantic foundation — something multi-agent system modeling is lacking.

We incorporate reasoning and learning into the model by refining the abstract model with Case-Based Reasoning. In so doing, we also develop formal executable semantics of an abstract case-based reasoner, with its subsequent refinement to a concrete CBR. This by itself deserves mention, since to the best of our knowledge there has been no prior attempt at deriving a formal specification of the Case-Based Reasoning Process.
We also present a novel Cognitive Navigation Algorithm, that incorporates a combination of exploration and learning, and takes into account person specific cognition of geography, in forming path planning decisions.

Mathematical and computational modeling of crime serves multiple purposes. It has a direct value in law enforcement, in intelligence led policing, and in proactive crime reduction and prevention. For intelligence led policing, this model would make it possible to predict likely activity space for serial offenders for precautions and for apprehension. For proactive policing, modeling of crime makes it feasible to build scenarios in crime analysis and prevention, and provides a basis for experimental research allowing experiments that can often not easily be done in the real world.

The particular challenge we face is the complexity and diversity of the problem space due to two major factors: a) the inherent complexity and dynamics of social systems, and b) the cross-disciplinary nature of the research field spanning Criminology, Environment Planning, Modeling & Simulation, AI, Navigation.
To this end, we can say that the ASM abstraction mechanisms greatly simplified the task of extracting and formalizing behavioral aspects of the system under study and were invaluable for delineating the borderline between the system and its operational environment.

Our main theoretical result is the abstract behavioral model of person agents interacting with their objective and subjective environments, and potentially with each other. Specifically, the model carves out the activity, awareness and crime occurrence space of criminal agents depending on their personal preferences. Although abstract, the model is complete with respect to the given level of detail. Our main practical result is an executable version of the distributed ASM model which is based on AsmL and is used for experimental validation of the abstract model. The AsmL model also serves as a platform for the construction of discrete event simulation models.

The model is designed for robustness with the intention to extend and refine it as required

to gradually incorporate principles and techniques from other research disciplines, e.g., for dealing with various cognitive aspects, especially in modeling the target selection process. Although the current model focuses on physical crime in urban environments, the model is general and abstract, and thus in principle scalable not only to different modes of crime, but different levels of spatial aggregation. It can thus be used to simulate a broad range of crimes — mundane crime like robbery, car theft, burglary etc.; crimes of passion such as serial murder, homicide, rape etc; non-conventional crime like corporate crime, cyber crime, intrusion detection. It can also be used at different levels of spatial aggregation viz, micro-level (airports, malls, downtown), meso-level (within cities, provinces) and macro-level (between countries, geopolitical crime).

Chapter 13

Opportunities and Future Research

As mentioned before, to the best of our knowledge, this work forms the first published research in Computational Criminology [19] [20] [18],[21] . In this vein, the abstract framework serves as a backbone for extending the model along different dimensions of refinement, for its full composition. This opens avenues for incorporating various problem-solving and decision-making techniques spanning multiple fields, that fill the niche space created by the abstract functions of the DASM model.

To begin with, various abstract functions of the Target Selection Module need to be refined. These functions will typically represent a combination of Case-Based Reasoning and Model-Based Reasoning, similar to the reasoning approach adopted in SEM. This entails deriving a concrete CBR from the abstract CBR for target selection; and also formalizing the patterns in target templating and selection, which needs to be carried out in consultation with the criminologists.

As highlighted in [56], Data Mining has great potential in Computational Criminology. Pertinent to our model, classification, clustering and feature selection algorithms commonly used in data mining [60] can be used for instantiating agents with their daily schedules and their personal preferences (profile). Daily schedules and preferences are typically based on demographic factors (age, race, gender) and socio-economic factors (income, race etc), and play a paramount role in determining the awareness space and subsequently the crime occurrence space of agents. These algorithms can be applied on data sets of criminals and their profiles. The School of Criminology at SFU has access to such databases, and it is

anticipated that the model will be integrated with such techniques in near future.

Probabilistic extensions of the model would help in supplementing predictive modeling. Since it is nearly impossible to attain a cent percent predictive power, probabilistic guesses are a better estimate for some scenarios. Specifically, by using a probabilistic technique in ADM, one can assign probabilities to the 'next destination' a person is likely to visit. Similarly, in the TSM, one can associate probabilities for the victimization of targets.

A mature visualization developed using sophisticated GIS applications such as ArcGis, MapInfo etc, and event-driven programming, with active user interaction would greatly enhance the analytics and understanding of the dynamics of the system. It would also supplement in carrying out further verification and validation of the model. We use a modest visualization developed in C++ and OpenGL by the Graphics Lab at SFU for the purpose of this thesis. However, further collaboration is anticipated with the Graphics Lab on this front.

One might pose the lack of substantial numerical results as a critique to measure the reliability of the model. However, as highlighted in Chapter 11, the problem of experimental validation is more daunting than it seems. In descriptive modeling, the problem of providing evidence for the validity of a model has a different meaning than in prescriptive modeling. A combination of qualititive and quantitative methods are called for. Developing effective verification and validation techniques deserves a more thorough investigation and is subject of our future work.

Finally, a note on the scalability of the model is noteworthy. The model is scalable with respect to two aspects (1) different modes of crime, and (2) different levels of spatial aggregation. Although currently we focus on physical crime in urban areas which includes a broad range of crimes — mundane crime like robbery, car theft, burglary etc.; crimes of passion such as serial murder, homicide, rape etc. — principles of environmental criminology suggest that that the theories embody the essence of other non conventional crimes as well. In principle, the model can be extended to simulate patterns of non-conventional crime like corporate crime, cyber crime, intrusion detection. It can also be applied to simulate patterns at different levels of spatial aggregation viz, micro-level (airports, malls, downtown), meso-level (within cities, provinces) and macro-level (between countries, geopolitical crime). This line of contention, however, needs more thorough analytical research, and is part of our future work.

APPENDICES

Appendix A

Abstract ASM Model

A.1 Global Definitions

A.1.1 Environment

Level 0 : Environment Graph

// ——— The Environment Graph ———
domain ENVIRONMENT_GRAPH
domain NODE
domain EDGE

// ——— Operations w.r.t the graph structure ———
nodeSet : ENVIRONMENT_GRAPH → NODE – *set*
edgeSet : ENVIRONMENT_GRAPH → EDGE – *set*

// ——— Node ———
outIncidentEdges : NODE × ENVIRONMENT_GRAPH → EDGE – *set*
adjacent : NODE × NODE × ENVIRONMENT_GRAPH → BOOLEAN

// ——— Edge ———
edgeHead : EDGE → NODE
edgeTail : EDGE → NODE

Level 1: Geographic Environment

// ——————— Geo Node Attributes ———————
domain GEO_STAT_NODE_ATTR // Θ_v^{stat}
domain GEO_DYN_NODE_ATTR // Θ_v^{dyn}
// $\Theta_v = (\Theta_v^{stat}, \Theta_v^{dyn})$
domain GEO_NODE_ATTR $\equiv$ GEO_STAT_NODE_ATTR $\cup$ GEO_DYN_NODE_ATTR

// ——————— Geo Edge Attributes ———————
domain GEO_STAT_EDGE_ATTR // Θ_e^{stat}
domain GEO_DYN_EDGE_ATTR // Θ_e^{dyn}
// $\Theta_e = (\Theta_e^{stat}, \Theta_e^{dyn})$
domain GEO_EDGE_ATTR $\equiv$ GEO_STAT_EDGE_ATTR $\cup$ GEO_DYN_EDGE_ATTR
// $\Theta = (\Theta_e, \Theta_v)$
domain GEO_ATTR $\equiv$ GEO_EDGE_ATTR $\cup$ GEO_NODE_ATTR
// ——————— Geographic Environment ———————
// $G_{GeoEnv} = (H, \theta)$
GEO_ENV $\equiv$ ENVIRONMENT_GRAPH *where*
geoAttr : GEO_ENV $\rightarrow$ GEO_ATTR

// Geo Environment functions
// $\theta_v^{stat} : V \rightarrow \mathcal{P}(\Theta_v^{stat})$
geoStaticNodeAttr : NODE $\times$ GEO_ENV $\times$ GEO_STAT_NODE_ATTR $\rightarrow$ VALUE
// $\theta_v^{stat} : V \rightarrow \mathcal{P}(\Theta_v^{dyn})$
geoDynamicNodeAttr : NODE $\times$ GEO_ENV $\times$ GEO_DYN_NODE_ATTR $\rightarrow$ VALUE
// $\theta_e^{stat} : E \rightarrow \mathcal{P}(\Theta_e^{stat})$
geoStaticEdgeAttr : EDGE $\times$ GEO_ENV $\times$ GEO_STAT_EDGE_ATTR $\rightarrow$ VALUE
// $\theta_e^{dyn} : E \rightarrow \mathcal{P}(\Theta_e^{dyn})$
geoDynamicEdgeAttr : EDGE $\times$ GEO_ENV $\times$ GEO_DYN_EDGE_ATTR $\rightarrow$ VALUE

Level 2: Subjective Environment

// ——————— Subjective Environment Attributes ———————
domain PER_EDGE_ATTR // λ_e^{per}
domain PER_NODE_ATTR // λ_v^{per}
domain PER_ATTR ≡ PER_EDGE_ATTR ∪ PER_NODE_ATTR // $\lambda_e^{per} = (\lambda_v^{per}, \lambda_e^{per})$

domain AW_EDGE_ATTR // λ_e^{aw}
domain AW_NODE_ATTR // λ_v^{aw}
domain AW_ATTR ≡ AW_EDGE_ATTR ∪ AW_NODE_ATTR // $\lambda_{AW} = (\lambda_v^{aw}, \lambda_e^{aw})$

domain AC_EDGE_ATTR // λ_e^{ac}
domain AC_NODE_ATTR // λ_v^{ac}
domain AC_ATTR ≡ AC_EDGE_ATTR ∪ AC_NODE_ATTR // $\lambda_{AC} = (\lambda_v^{ac}, \lambda_e^{ac})$

domain SUBJ_ATTR ≡ PER_ATTR ∪ AW_ATTR ∪ AC_ATTR
domain SUBJ_EDGE_ATTR ≡ PER_EDGE_ATTR ∪ AW_EDGE_ATTR ∪ AC_EDGE_ATTR
domain SUBJ_NODE_ATTR ≡ PER_NODE_ATTR ∪ AW_NODE_ATTR ∪ AC_NODE_ATTR

// ——————— Subjective Environment ———————
// $G_{SubEnv} = (G_{GeoEnv}, \mathrm{A})$
SUBJ_ENV ≡ GEO_ENV *where*
subjAttr : PERSON × SUBJ_ENV → SUBJ_ATTR

// Subjective Environment Related Functions
// A_e : AGENT × $E \rightharpoonup \mathcal{P}(\lambda_e)$
subjEdgeAttr : PERSON × EDGE × SUBJ_ENV × SUBJ_EDGE_ATTR → VALUE
// A_v : AGENT × $V \rightharpoonup \mathcal{P}(\lambda_v)$
subjNodeAttr : PERSON × NODE × SUBJ_ENV × SUBJ_NODE_ATTR → VALUE

Level 3: Refined Attributes

// ——————— Geo Static Node Attribute ———————
coordinate : → GEO_STAT_NODE_ATTR
nodeName : → GEO_STAT_NODE_ATTR

// ——————— Geo Static Edge Attribute ———————
distance : → GEO_STAT_EDGE_ATTR
roadType : → GEO_STAT_EDGE_ATTR
edgeName : → GEO_STAT_EDGE_ATTR

// ——————— Geo Dynamic Node Attribute ———————
traffic : → GEO_DYN_EDGE_ATTR
roadCondition : → GEO_DYN_EDGE_ATTR

// ——————— Perception Attributes ———————
PER_EDGE_ATTR ≡ GEO_EDGE_ATTR // λ_e^{per}
PER_NODE_ATTRRIBUTE ≡ GEO_NODE_ATTR // λ_v^{per}

// ——————— Awareness Attributes ———————
AW_EDGE_ATTR : → *intensity*

// derived function for awareness space
awarenessSpace : PERSON × SUBJ_ENV → EDGE − *Set*
activeAwarenessSpace : PERSON × SUBJ_ENV × VALUE → EDGE − *Set*

// ——————— Activity Attributes ———————
frequency :→ AC_EDGE_ATTR
tripImportance :→ AC_EDGE_ATTR
reinforcement :→ AC_EDGE_ATTR

// derived function for activity space
activitySpace : PERSON × SUBJ_ENV → EDGE -- *Set*
activeActivitySpace : PERSON × SUBJ_ENV × VALUE → EDGE − *Set*

Level 4: Refining the abstract domain VALUE

VALUE ≡ REINFORCEMENT ∪ TRIP_IMPORTANCE ∪ FREQUENCY
INTENSITY ∪ TRAFFIC ∪ DISTANCE∪
ROAD_TYPE ∪ ROAD_CONDITION
DISTANCE ≡ INTENSITY ≡ FLOAT
FREQUENCY ≡ INTEGER

// ——————— REINFORCEMENT ———————
positive :→ REINFORCEMENT
negative :→ REINFORCEMENT
neutral :→ REINFORCEMENT

// ——————— TRIP_IMPORTANCE ———————
obligatory :→ TRIP_IMPORTANCE
required :→ TRIP_IMPORTANCE
notRequired :→ TRIP_IMPORTANCE

// ——————— TRAFFIC ———————
low :→ TRAFFIC
medium :→ TRAFFIC
high :→ TRAFFIC
blocked :→ TRAFFIC

// ——————— ROAD_TYPE ———————
minor :→ ROAD_TYPE
major :→ ROAD_TYPE
highway :→ ROAD_TYPE

// ——————— ROAD_CONDITION ———————
favorable :→ ROAD_CONDITION
unfavorable :→ ROAD_CONDITION
neutral :→ ROAD_CONDITION

A.1.2 Linking Social Systems to DASM Models

// ——————— Mapping ———————
// Some Social Entities
domain COP, CRIMINAL
domain CAR, BUS
domain DRUGS, CASH

// MAS Entities
domain ENTITY
domain PASSIVE_OBJECT
domain ACTIVE_OBJECT
domain AUTONOMOUS_AGENT
ENTITY ≡ PASSIVE_OBJECT ∪ ACTIVE_OBJECT ∪ AUTONOMOUS_AGENT

// DASM Entities
domain AGENT

// Mapping
PASSIVE_OBJECT ≡ DRUGS ∪ CASH
ACTIVE_OBJECT ≡ CAR ∪ BUS
AUTONOMOUS_AGENT ≡ COP ∪ CRIMINAL
AGENT ≡ AUTONOMOUS_AGENT

// ——————— Hierarchical Classification ———————
domain ATTRIBUTES
domain BEHAVIOR
domain RULES
domain MEMORY
domain MOTIVATIONS

// ——————————Passive Object——————————
static *attributes* : PASSIVE_OBJECT → ATTRIBUTES

// ——————————Active Object——————————
ACTIVE_OBJECT ≡ PASSIVE_OBJECT *where*

behavior : ACTIVE_OBJECT → BEHAVIOR
behavior(*a*) ≡ **dynamic** *attributes*

// ——————————Autonomous Agent——————————
AUTONOMOUS_AGENT ≡ ACTIVE_OBJECT *where*

rules : AUTONOMOUS_AGENT → RULES
rules(*a*) ≡ *Program*(*a*)

memory : AUTONOMOUS_AGENT → MEMORY

motivations : AUTONOMOUS_AGENT → MOTIVATION – *Set*

It should be noted that the formal representation of the classification presented here is used as a means for understanding the system. While we use explicit domains here for MEMORY, RULES, BEHAVIOR, ATTRIBUTES, in the subsequent instantiation of autonomous agents, we don't extend these domains vis-a-vis. Instead, we may use abstract and derived functions, and other domains, that constitute these domains indirectly.

A.1.3 Person Agent

Level 0

domain PERSON
domain MODULE
domain AGENT

domain SEM
domain TSM
domain ADM
MODULE ≡ SEM ∪ TSM ∪ ADM

AGENT ≡ PERSON ∪ MODULE

// ———————— The environment ————————
monitored *geoEnv* : → GEO_ENV

// ———————— Modules ————————
parentAgent : MODULE → PERSON

// ———————— Person ————————
spaceModule : PERSON → SEM
targetModule : PERSON → TSM
decisionModule : PERSON → ADM

// Auxiliary data structures
currentNode : PERSON → NODE
currentEdge : PERSON → EDGE

// ———————— Person Agent Program ————————
PERSON_Program
SEM_Program
TSM_Program
ADM_Program

Level 1

// ———————— CBR Components of the Modules. ————————
domain CBR
domain SEM_CBR
domain TSM_CBR
CBR ≡ SEM_CBR ∪ TSM_CBR

// Respective CBR components
semCBR : SEM → SEM_CBR
semCBRDominant : SEM → BOOLEAN // Determines if Memory or Explorer is dominant.

tsmCBR : TSM → TSM_CBR
tsmCBRDominant : TSM → BOOLEAN // Determines if Memory or Explorer is dominant.

A.1.4 Signals

// ——————— Signals Triggered by SEM ———————
domain INFORM_ARRIVAL
// Triggered to ADM to inform person has reached its current destination.
domain NEW_PROBLEM
// Triggered to CBR to give a path from given source to destination node.
domain FEEDBACK_AVAILABLE
// Triggered to CBR when feedback for a proposed path is available.
// i.e, when the person has reached its destination.

// ——————— Signals Triggered by ADM ———————
domain NEW_DEST
// Triggered to SEM, when a new destination is available to be traveled to.
domain WEIGHTS_UPDATED
// Triggered to SEM, when the weights of influence factors get changed.
domain CRIMINAL_MOTIVATED
// Triggered to TSM, when the person's criminal propensity is above a threshold.

// ——————— Signals Triggered by CBR ———————
domain PROBLEM_SOLVED
// Triggered to its owner(SEM), when the given problem has been solved.
domain INIT
// Triggered to its Post Solution Module, to start initialization.

// ——————— Signals Triggered by TSM ———————
domain TARGET_SELECTED
// Triggered to ADM, to inform which targets are selected.

// ——————— Signals ———————
SIGNAL ≡
INFORM_ARRIVAL ∪ NEW_PROBLEM ∪ FEEDBACK_AVAILABLE
NEW_DEST ∪ WEIGHTS_UPDATED ∪ CRIMINAL_MOTIVATED
PROBLEM_SOLVED ∪ INIT

The structure of each SIGNAL is formulated in the corresponding level of the model it is used in. However, for sake of simplicity, we enumerate below the structure of all the signals. For details on the keywords 'Trigger' and 'OnSignal', the reader is referred to [41].

// ——————— INFORM_ARRIVAL ———————
arrivalTime : INFORM_ARRIVAL → TIME
path : INFROM_ARRIVAL → PATH

// ——————— NEW_PROBLEM ———————
problem : NEW_PROBLEM → PROBLEM
owner : NEW_PROBLEM → SEM

// ——————— FEEDBACK_AVAILABLE ———————
cbrProblem : FEEDBACK_AVAILABLE → PROBLEM
cbrSolution : FEEDBACK_AVAILABLE → SOLUTION
externalFeedback : FEEDBACK_AVAILABLE → FEEDBACK

// ——————— NEW_DEST ———————
newDest : NEW_DEST → NODE

// ——————— PROBLEM_SOLVED ———————
solution : PROBLEM_SOLVED → PATH

A.2 Space Evolution Module (SEM)

A.2.1 Level 0

Definitions

domain PATH $\equiv$ NODE – *Seq*

domain MODE $\equiv \{idle, pathPlanning, roadSelection, localRePlanning, running, pathCompleted\}$

// ———————— Structure of Signal NEW_DEST ————————
newDest : NEW_DEST → NODE

// ———————— Kept in Working Memory ————————
mode : SEM → MODE // SEM has a mode.

destNode : SEM → NODE // Current destination node.

sourceNode : SEM → NODE // The source node.

currentEdge : SEM → NODE // Current edge stored in 'Person' Agent.
currentEdge(*a*) $\equiv$ *currentEdge*(*parentAgent*(*a*))

currentNode : SEM → NODE // Current node stored in 'Person' Agent.
currentNode(*a*) $\equiv$ *currentNode*(*parentAgent*(*a*))

// Stores the path suggested by the algorithm for traversal.
suggestedPath : SEM → PATH
// Stores the edge of the 'suggested path' suggested for immediate traversal.
suggestedEdge : SEM → EDGE
// Stores the final path the agent actually traverses.
takenPath : SEM → PATH

// Decides whether 'suggestedEdge' is fit for traversal, based on current road conditions.
acceptable : SEM × EDGE → BOOLEAN
// Decides whether the time required for traveling along the current edge has elapsed.
currentEdgeTraversed : SEM → BOOLEAN
// Decides whether the agent has reached the destinaitn node.
destNodeReached : SEM → BOOLEAN
destNodeReached $\equiv$ *currentNode*(*self*) = *destNode*(*self*)

Rules

```
// ———————— Space Module Program ————————
SEM_Program ≡
  case mode of
    idle →
      onsignal s : NEW_DEST
        INITIALIZE(currentNode, newDest(s))
        mode := pathPlanning

    pathPlanning →
      GET_PATH   // suggestedPath is updated
      mode := roadSelection

    roadSelection →
      if destNodeReached then
        mode := pathCompleted
      else
        if signalFromADM then
          HANDLE_ADM_SIGNALS
          mode := pathPlanning
        else
          GET_SUGGESTED_EDGE   // suggestedEdge is updated
          mode := localRePlanning

    localRePlanning →
      if acceptable(suggestedEdge) then
        mode := running
        currentEdge := suggestedEdge
      else
        mode := pathPlanning
      RECORD_SELECTED_EDGE

    running →
      if currentEdgeTraversed then
        UPDATE_EDGE_PERCEPTION
        FINALIZE_EDGE_TRAVERSAL
        SET_SEM_MODE

    pathCompleted →
      FINALIZE_TRIP
      mode := idle
where
  currentEdge ≡ currentEdge(parentAgent(self))
  destNodeReached ≡ currentNode(parentAgent(self)) = destNode(self)
```

A.2.2 Level 1

Definitions

// ——————— Structure of Signal INFORM_ARRIVAL ———————
arrivalTime : INFORM_ARRIVAL → TIME
path : INFROM_ARRIVAL → PATH

// ——————— Kept in Profile ———————
cbrDominant : SEM → BOOLEAN
cbrDominant(*sem*) ≡ *semCBRDominant*(*parentAgent*(*sem*))

// ——————— Kept in Working Memory ———————
readyToExplore : SEM → BOOLEAN
// if true, explorer is ready to explore

randomEdgeSelected : SEM → BOOLEAN

attemptedEdge : SEM → EDGE

traverseStartTime : SEM → TIME

acceptable(*sem*, *e*) ≡ *attemptedEdge*(*sem*) = *e* ∨ *goodLocalCondition*(*sem*, *e*)

goRandom : SEM → BOOLEAN
// Determines whether the agent takes a random edge

waitingForSignal : SEM → BOOLEAN
// Determines whether the agent is waiting for a solution from the CBR

Rules

INITIALIZE(*source* : NODE, *dest* : NODE) ≡
  destNode := *dest*
  sourceNode := *source*
  takenPath := {*source*}

GET_PATH ≡
  if *cbrDominant* **then**
    Get_Suggested_Path$_{Memory}$
  else
    Get_Suggested_Path$_{Explorer}$

Get_Suggested_Path$_{Explorer} \equiv$
  $suggestedPath \leftarrow$ GET_SUGESTED_PATH$_{Explorer}(self, currentNode, destNode)$

Get_Suggested_Path$_{Memory} \equiv$
  let $pathCBR \leftarrow$ GET_SUGGESTED_PATH$_{CBR}(currentNode, destNode)$ **in**
    if $\neg empty(pathCBR) \wedge newPath(pathCBR)$ **then**
      if $complete(pathCBR)$ **then**
        $suggestedPath := pathCBR$
      if $\neg complete(pathCBR)$ **then**
        $suggestedPath \leftarrow$ GET_SUGGESTED_PATH$_{Mixed}(pathCBR)$
    if $empty(pathCBR) \vee \neg newPath(pathCBR)$ **then**
      $suggestedPath \leftarrow$ GET_SUGESTED_PATH$_{Explorer}(self, currentNode, destNode)$
where
  $complete(p) \equiv tail(p) = destNode$
  $newPath(pathCBR) \equiv pathCBR \neg \subseteq suggedtedPath(self)$

HANDLE_ADM_SIGNALS $\equiv$
  onsignal s : NEW_DEST
    INITIALIZE$(currentNode, newDest(s))$
  onsignal s : WEIGHTS_UPDATED
    UPDATE_WEIGHTS(s)

GET_SUGGESTED_EDGE $\equiv$
  if $goRandom$ **then**
    Get_Suggested_Edge$_{Path}$
  if $\neq goRandom$ **then**
    Get_Suggested_Edge$_{Random}$

Get_Suggested_Edge$_{Random} \equiv$
    choose e **in** $outIncidentEdges(currentNode)$
      $suggestedEdge := e$
      $randomEdgeSelected := true$

Get_Suggested_Edge$_{Path} \equiv$
    let $edge = firstEdge(suggestedPath)$ **in**
      $suggestedEdge := edge$
      remove $head(suggestedPath)$ **from** $suggestedPath$
where
  $firstEdge(p) \equiv edge(head(p), second(p))$

RECORD_SELECTED_EDGE ≡
  if *acceptable*(*suggestedEdge*) **then**
    currentNode := *undef*
    // the agent starts passing the edge
    traverseStartTime := *now*
  else
    attemptedEdge := *suggestedEdge*

FINALIZE_EDGE_TRAVERSAL ≡
  let *node* = *edgeTail*(*currentEdge*) **in**
    currentNode := *node*
    currentEdge := *undef*
    add *node* **to** *takenPath*

SET_SEM_MODE ≡
  if ¬*randomEdgeSelected* **then**
    mode := *roadSelection*
  else
    randomEdgeSelected := *false*
    mode := *pathPlanning*

FINALIZE_TRIP ≡
  trigger *s* : INFORM_ARRIVAL, *decisionModule*(*parentAgent*)
    arrivalTime(*s*) := *now*
    path(*s*) := *takenPath*
  SEND_FEEDBACK_TO_CBR(*curProblem*, *pathCBR*, *takenPath*)
  readyToExplore := *false*

A.2.3 Level 2

Definitions

// ——————— Structure of Signal PROBLEM_SOLVED. ———————
solution : PROBLEM_SOLVED → PATH

// ——————— Kept in Working Memory ———————
curProblem : SEM → PROBLEM
closeness : SEM → INTEGER

Rules

GET_SUGGESTED_PATH$_{CBR}$(*currentNode* : NODE, *destNode* : NODE) ≡
  if ¬*waitingForSignal* **then**
    SEND_NEW_PROBLEM_TO_CBR(*currentNode*, *destNode*, *closeness*)
    waitingForSignal := *true*
  if *waitingForSignal* **then**
    onsignal *s* : PROBLEM_SOLVED
      return *solution*(*s*)
      waitingForSignal := *false*
where
  closeness = *closeness*(*self*)

GET_SUGGESTED_PATH$_{Mixed}$(*partialPathCBR* : PATH) ≡
 let *partialPathExplorer* ← GET_SUGGESTED_PATH$_{Explorer}$(*self*, *tail*(*partialPathCBR*), *destNode*)
 in
  let
    pathMixed ← *concat*(*partialPathCBR*, *partialPathExplorer*)
    explorerPath ← GET_SUGGESTED_PATH$_{Explorer}$(*self*, *currentNode*, *destNode*)
   in
    return *superior*(*pathMixed*, *pathExplorer*)

A.2.4 Level 3

Definitions

domain PROBLEM
domain FEEDBACK
domain SOLUTION ≡ PATH

// ——— Structure of Signal NEW_PROBLEM ———
problem : NEW_PROBLEM → PROBLEM
owner : NEW_PROBLEM → SEM

// ——— Structure of Signal FEEDBACK_AVAILABLE ———
cbrProblem : FEEDBACK_AVAILABLE → PROBLEM
cbrSolution : FEEDBACK_AVAILABLE → SOLUTION
externalFeedback : FEEDBACK_AVAILABLE → FEEDBACK

// ——————— PROBLEM ———————
source : PROBLEM → NODE
dest : PROBLEM → NODE
time : PROBLEM → TIME
closeness : PROBLEM → INTEGER

// ——————— FEEDBACK ———————
pathTaken : FEEDBACK → PATH

Rules

SEND_NEW_PROBLEM_TO_CBR(*source* : NODE, *dest* : NODE, *closeness* : INTEGER) ≡
  let p = CREATE_NEW_PROBLEM(*source*, *dest*, *now*, *closeness*) **in**
    trigger s : NEW_PROBLEM, $semCBR$
      problem(s) := p
      owner(s) := *self*
    SEND_FEEDBACK_FOR_PARTIAL_SOLUTION
    curProblem := p

SEND_FEEDBACK_TO_CBR(p : PROBLEM, *sol* : SOLUTION, tp : PATH) ≡
  trigger s : FEEDBACK_AVAILABLE, $semCBR$
    $cbrSolution(s)$:= *sol*
    $cbrProblem(s)$:= p
    $externalFeedback(s)$ ← CREATE_FEEDBACK($takenPath$)

SEND_FEEDBACK_FOR_PARTIAL_SOLUTION ≡
  if *abandonedSolution* **then**
    SEND_FEEDBACK_TO_CBR(*curProblem*, $pathCBR$, *undef*)
    // Sending *undef* as feedback implies the path was abandoned.
where
  abandonedSolution ≡ *dest*(*curProblem*) ≠ *currentNode*

A.3 Target Selection Module (TSM)

A.3.1 Level 0

Definitions

// ——————— Domains for the TSM ———————
domain MODE ≡ {*observing*, *targetTemplating*, *targetSelection*}

domain CRIME_TYPE
carTheft :→ CRIME_TYPE
shopLift :→ CRIME_TYPE
robbery :→ CRIME_TYPE

// Target
domain LOCATION ≡ NODE ∪ EDGE
domain TARGET ≡ PASSIVE_OBJECT ∪ ACTIVE_OBJECT

crimeType : TARGET → CRIME_TYPE
location : TARGET → LOCATION

// Functions added to Edge and Node
potentialTargets : EDGE → TARGET − *set*
potentialTargets : NODE → TARGET − *set*

// ——————— Kept in Profile ———————
criminality : TSM × CRIME_TYPE → PROBABILITY
// Specialty of the criminal for diff. crimes

// ——————— Kept in Working Memory ———————
mode : TSM → MODE // TSM has a mode

potentialTargets : TSM → TARGET − *Set* // All observed targets.
goodTargets : TSM → TARGET − *Set* // Targets 'templated' as 'good'
selectedTargets : TSM → TARGET − *Set* // Targets eventually victimized
currentLocation : TSM → LOCATION // Current node or edge of the person

Rules

```
// ———————— TSM Program ————————
TSM_Program ≡
  case mode of
    observing →
      onsignal s : CRIMINAL_MOTIVATED
        if currentLocation ≠ undef then
          // Get all the targets located 'around, on' current Location.
          GET_POTENTIAL_TARGETS(currentLocation)// Sets potentialTargets.
          mode := targetTemplating

    targetTemplating →
      // Compare environmental cues of target against your own template
      // Based on templating, categorise targets as 'good' or 'bad'
      // This process can be based on memory/cases(CBR) or explicitly executed(MBR)
      TARGET_TEMPLATING(potentialTargets)// Sets goodTargets.
      mode := targetSelection

    targetSelection →
      // Victimize good targets, reinforce target template.
      SELECT_TARGETS(goodTargets)// Sets selectedTargets
      mode := observing
```

A.4 Agent Decision Module (ADM)

A.4.1 Level 0

Definitions

// ——————— Domains ———————
// Motivation
routineActivity :→ MOTIVATION
criminalPropensity :→ MOTIVATION

// ——————— Kept in Profile ———————
motivations : ADM → MOTIVATION – *Set* // Person has motivations
threshold : ADM × MOTIVATION → VALUE // Motivation threshold to activate behavior

Rules

ADM_Program ≡
if ROUTINE_ACTIVITY(*routineActivity*, *self*) ≥ *threshold*(*routineActivity*)
then
ADM_SEM_MONITOR
if CRIMINAL_PROPENSITY(*criminalyPropensity*, *self*) ≥ *threshold*(*criminalyPropensity*)
then
ADM_TSM_MONITOR

A.4.2 Level 1 : ADM_SEM_MONITOR

Definitions

// ———————— Domains ————————
domain MODE ≡ {*monitor*, *calculate*, *decide*, *inform*}

domain TIME
domain DATE
domain DAY
domain TIMEOFDAY
domain DAYTYPE ≡ {*weekday*, *weekend*}

domain PROBABILITY

// –PROBABLE_DEST–
toNode : PROBABLE_DEST → NODE
prob : PROBABLE_DEST → PROBABILITY
// –APPOINTMENT–
timeOfDay : APPOINTMENT → TIMEOFDAY
toNodeSet : APPOINTMENT → PROBABLE_DEST – *Set*
// –SCHEDULE–
fromDate : SCHEDULE → DATE
toDate : SCHEDULE → DATE
dayType : SCHEDULE → DAYTYPE // weekday, weekend
dailySchedule : SCHEDULE → APPOINTMENT – *Set*
// –PERSONAL_SCHEDULE–
regularSchedule : PERSONAL_SCHEDULE → SCHEDULE – *Set* // fromDate = toDate = undef
specialSchedule : PERSONAL_SCHEDULE → SCHEDULE – *Set* // fromDate not undef

// ———————— Kept in Profile ————————
schedule : ADM → PERSONAL_SCHEDULE

// ———————— Kept in Working Memory. ————————
mode : ADM → MODE // mode of ADM
nextDest : ADM → NODE // next destination calculated
// —Auxiliary—
arrived : ADM → BOOLEAN // whether the person has reached its current destination
timeChanged : ADM → BOOLEAN // time has changed
lastTime : ADM → TIMEOFDAY // records last time
inform : ADM → BOOLEAN // inform the SEM?

Rules

```
// ———————— ADM_SEM_MONITOR ————————
ADM_SEM_MONITOR ≡
  case mode of
    monitor →
      // Record the fact agent has arrived at a dest.
      onsignal s : INFORM_ARRIVAL
        arrived(self) := true
      MONITOR_TIME(self)
      // Keep monitoring time to see if it changes(ex: morning to noon, 3pm to 4pm.)
      // If changes, set mode to calculate.
    seq
    calculate →
      // Probabilistically choose the next destination in the schedule
      // Set nextDest.
      CALCULATE_NEXT_DEST(self)
      mode := decide
    seq
    decide →
      if nextDest(self) then
        // Decide whether to inform the agent or not - Set Inform predicate
        // Usually if arrived = true inform, else decide.
        DECIDE_TO_INFORM(self)
        mode := inform
      else
        // No schedule available. Go back to Monitoring time.
        mode := monitor
    seq
    inform →
      if inform(self) then
        // triggers the SEM with the New Destination.
        trigger s : NEW_DEST, spaceModule(parentAgent)
          newDest(s) := nextDest(self)
      mode := monitor
```

A.4.3 Level 1: ADM_TSM_MONITOR

Rules

```
// ———————— ADM_TSM_MONITOR ————————
ADM_TSM_MONITOR ≡
  // triggers the TSM to initialize its execution.
  trigger s : CRIMINALLY_MOTIVATED, targetModule(parentAgent)
```

A.4.4 Level 2

Definitions

domain HH
domain MM
domain MONTH
domain DAYOFMONTH
domain YEAR
domain TIMETYPE $\equiv \{morning, afternoon, evening\}$

// ——————— TIME ———————
$GetYear$: TIME $\rightarrow$ YEAR
$GetYear(t) \equiv t/(12 * 30 * 24 * 60)$ *mod* 1

$GetMonth$: TIME $\rightarrow$ MONTH
$GetMonth(t) \equiv t/(30 * 24 * 60)$ *mod* 12

$GetDay$: TIME $\rightarrow$ DAYOFMONTH
$GetDay(t) \equiv t/(24 * 60)$ *mod* 30

$GetHour$: TIME $\rightarrow$ HH
$GetHour(t) :\equiv (t/60)$ *mod* 24

$GetMinute$: TIME $\rightarrow$ MM
$GetMinute(t) :\equiv (t/1)$ *mod* 60

$GetDayType$: TIME $\rightarrow$ DAYTYPE
$$GetDayType(t) \equiv \begin{cases} weekday & : GetDay(t) \text{ mod } 7 \text{ in } \{0..4\} \\ weekend & : otherwise \end{cases}$$

Rules

// ——————— Monitor Time ———————
MONITOR_TIME(ADM : *self*) $\equiv$
  // Here, we monitor changes in time by the Hour
  let $currentHour = GetHour(now)$ **in**
    if $currentHour \neq lastTime$ **then**
      $timeChanged(self) := true$
      $lastTime(self) := currentHour$
      $mode := calculate$

```
// ———————— Calculate Next Dest ————————
CALCULATE_NEXT_DEST(ADM : self) ≡
  choose ss from specialSchedule(schedule) with matchss(ss)
    FindAppt(self, ss)
  if none
    choose rs from regularSchedule(schedule) with matchrs(rs)
      FindAppt(self, rs)
    if none
      nextDest(self) := undef
where
  matchss(ss) ≡ GetDate(now) ∈ date(ss) ∧ GetDay(now) = day(ss)
  matchrs(rs) ≡ GetDayType(now) = GetDayType(rs)
  date(ss) ≡ {fromDate(ss) − toDate(ss)}
  day(ss) ≡ dayType(ss) ∨ undef

FindAppt(self : ADM, s : SCHEDULE) ≡
  choose app from dailySchedule(s) with matcht(app)
    ChooseDest(self, app)
  if none
    nextDest(self) := undef
where
  matcht(app) ≡ GetHour(now) = GetHour(timeOfDay(app))

ChooseDest(ADM : self, app : APPOINTMENT) ≡
  choose d in toNodeSet(app) with PROB(d)
    nextDest(self) := toNode(d)
    // Sets newDest according to the probabilities assigned.

// ———————— Decide ————————
DECIDE_TO_INFORM(ADM : self) ≡
  if arrived(self) then
    inform(self) := true
  else
    // Ask the agent is it wants to change its destination.
    inform(self) := ASKAGENT(self)
```

Appendix B

Refined ASM Model

B.1 Case-Based Reasoner (CBR)

B.1.1 Abstract CBR: Level 0 - 3

Level 0

Definitions

domain CBR
domain POST_SOL_MODULE
domain OWNER
AGENT $\equiv$ CBR $\cup$ POST_SOL_MODULE $\cup$ OWNER

domain MODE $\equiv \{idle, retrieve, reuse, done\}$

domain CASE
domain CASE-Index
domain CASE-Content
domain CASE-Outcome

domain PROBLEM
domain SOLUTION

// ----------CASE----------
caseIndex : CASE $\rightarrow$ CASE-Index
caseContent : CASE $\rightarrow$ CASE-Content
caseOutcome : CASE $\rightarrow$ CASE-Outcome

// ----------CBR----------
mode : CBR → MODE
caseBase : CBR → CASE − *Set*
postSolModule : CBR → POST_SOL_MODULE
owner : CBR → OWNER

ballParkSolution : CBR → SOLUTION
finalSolution : CBR → SOLUTION
problem : CBR → PROBLEM

Rules

// ---------- CBR Program ----------
CBR_Program ≡
  case *mode* **of**
    idle →
      onsignal *s* : NEW_PROBLEM
        problem(*self*) := *problem*(*s*)
        owner(*self*) := *owner*(*s*)
        mode := *retrieve*

    retrieve →
      // This sets the ballParkSolution
      // This sets the mode to reuse
      RETRIEVE(*self*)

    reuse →
      // This sets the finalSolution
      // This sets the mode to done
      REUSE(*self*)

    done →
      // triggers the postSol Module to start running in parallel.
      trigger *s* : INIT, *postSolModule*(*self*)
      // Send th solution back to the owner.
      trigger *s* : PROBLEM_SOLVED, *owner*(*self*)// send to whom?
        solution(*s*) := *finalSolution*(*self*)
      mode := *idle*

where
  mode ≡ *mode*(*self*)

Level 1

Definitions

```
// –Used by Retrieve Rule–
domain RETRIEVE_MODE ≡ {idle, identify, match, rank}
retrieveMode : CBR → RETRIEVE_MODE
identifiedIndex : CBR → CASE-Index
matchedCases : CBR → CASE − Set

// –Used by Reuse Rule–
domain REUSE_MODE ≡ {idle, copy, adapt}
reuseMode : CBR → REUSE_MODE
isCopy : CBR → BOOLEAN
```

Rules

```
// ———————— RETRIEVE ————————
RETRIEVE(self : CBR) ≡
  case retrieveMode of
    idle →
      retrieveMode := identify

    identify →
      // This sets identifiedIndex
      IDENTIFY(self, problem(self))
      retrieveMode := match

    match →
      // This sets matchedCases, based on identifiedIndex
      MATCH(self, identifiedIndex(self))
      retrieveMode := rank

    rank →
      // This sets the ballparkSolution, based on matchedCases.
      RANK(self, matchedCases(self))
      retrieveMode := idle
      mode := reuse

where
  retrieveMode ≡ retrieveMode(self)
  mode ≡ mode(self)
```

```
// ———————— REUSE ————————
REUSE(self : CBR) ≡
  case reuseMode of
    idle →
      if isCopy(self) then
        reuseMode := copy
      else
        reuseMode := adapt

    copy →
      finalSolution(self) := ballparkSolution(self)
      // Sets finalSolution
      reuseMode := idle
      mode := done

    adapt →
      // This sets finalSolution, by adapting the ballpark Solution
      ADAPT(self, ballparkSolution(self))
      reuseMode := idle
      mode := done

where
  reuseMode ≡ reuseMode(self)
  mode ≡ mode(self)
```

Level 2 : Post Solution Module

Definitions

```
domain POST_SOL_MODE ≡ {idle, evaluate, retain, done}
domain FEEDBACK
domain UNEVAL_CASE
domain EXTRACTED_INFO

// ———————UNEVALUATED_CASE———————-
unevalProblem : UNEVAL_CASE → PROBLEM
unevalSolution : UNEVAL_CASE → SOLUTION
feedback : UNEVAL_CASE → FEEDBACK
repairedSolution : UNEVAL_CASE → SOLUTION
extractedInfo : UNEVAL_CASE → EXTRACTED_INFO
```

// ——————POST_SOL_MODULE——————-
mode : POST_SOL_MODULE → POST_SOL_MODE
parentCBR : POST_SOL_MODULE → CBR
unevalCaseSet : POST_SOL_MODULE → UNEVAL_CASE – *Set*

// –Auxiliary–
unevalCase : POST_SOL_MODULE → UNEVAL_CASE
addAsUnevalCase : POST_SOL_MODULE → BOOLEAN
integrateFeedback : POST_SOL_MODULE → BOOLEAN

Rules

```
// ———————— PostSolutionModule Program ————————
POST_SOL_MODULE_Program ≡
  case mode of
    idle →
      onsignal s : INIT
        if addAsUnevalCase(self) then
          extend UNEVAL_CASE with newcase
            unevalProblem(newCase) := problem(parentCBR(self))
            unevalSolution(newCase) := finalSolution(parentCBR(self))
            add newCase to unevalCaseSet(self)

      onsignal s : FEEDBACK_AVAILABLE
        choose x in unevalCaseSet(self) with match(x, s)
          unevalCase(self) := x
          feedback(x) := externalFeedback(s)
          // Sets unevalCase with x and its feedback from environment.
        if none
          extend UNEVAL_CASE with unevalCase
            // Fill associated info into unevalCase.
            unevalProblem(unevalCase) := cbrProblem(s)
            unevalSolution(unevalCase) := cbrSolution(s)
            feedback(unevalCase) := externalFeedback(s)
            add unevalCase to unevalCaseSet(self)
        if integrateFeedback(s) then
          psMode := evaluate
        else
          psMode := done

    evaluate →
      // Sets the repairedSolution.
      // Sets the mode to Retain.
      EVALUATE(self)

    retain →
      // Sets the mode to done.
      RETAIN(self)

    done →
      remove unevalCase(self) from unevalCaseSet(self)
      mode := idle

where
  mode ≡ mode(self)
  match(x, s) ≡ unevalProblem(x) = problem(s) ∧ unevalSolution(x) = solution(s)
```

Level 3

Definitions

domain EVALUATE_MODE $\equiv \{idle, analyze, repair\}$
domain RETAIN_MODE $\equiv \{idle, extract, integrate\}$

// –Functions used by Evaluate–
$evaluateMode$: POST_SOL_MODULE $\rightarrow$ EVALUATE_MODE
$repairedNeeded$: POST_SOL_MODULE $\rightarrow$ BOOLEAN

// –Functions used by Retain–
$retainMode$: POST_SOL_MODULE $\rightarrow$ RETAIN_MODE

Rules

```
// ———————— EVALUATE ————————
EVALUATE(self : POST_SOL_MODULE) ≡
  case evaluateMode of
    idle →
      evaluateMode := analyze

    analyze →
      // Sets repairNeeded predicate.
      ANALYZE(self, unevalCase)
      evaluateMode := check

    check →
      // Based on feedback, Ananlyze the problem, solution to see if repair needed.
      if repairNeeded(self) then
          evaluateMode := repair
      else
          unevalSolution(unevalCase) := undef
          repairedSolution(unevalCase) := unevalSolution(unevalCase)
          // set the confirmed solution
          evaluateMode := idle
          mode := retain

    repair →
      // Sets the unevalSolution(unevalCase).
      REPAIR(self, unevalCase)
      evaluateMode := analyze
      // Based on feedback, repair the solution recursively until satisfied.

where
  evaluateMode ≡ evaluateMode(self)
  mode ≡ mode(self)
  unevalCase ≡ unevalCase(self)
```

```
// ———————— RETAIN ————————
RETAIN(self : POST_SOL_MODULE) ≡
  case retainMode of
    idle →
      retainMode := extract

    extract →
      // Sets extractedInfo.
      EXTRACT(self, unevalCase)
      retainMode := integrate

    integrate →
      // Integrates the extracted to the CaseBase.
      INTEGRATE(self, extractedInfo(unevalCase))
      retainMode := idle
      mode := done

where
  retainMode ≡ retainMode(self)
  mode ≡ mode(self)
```

B.1.2 Concrete CBR : Level 4

Level 1 - Concrete Case-Based Reasoner

Definitions

domain TIMETYPE ≡ {*morning*, *afternoon*, *evening*}
domain TIME
domain TIMEOFDAY
domain DATE

domain VALUE
domain WEIGHT

domain SEM_CBR ≡ CBR *where*
domain PATH ≡ NODE − *Seq*
SOLUTION ≡ NODE − *Seq*
OWNER ≡ SEM
EXTRACTED_INFO ≡ CASE − *Set*

// ——————TIME——————
date : TIME → DATE
timeOfDay : TIME → TIMEOFDAY
timeType : TIME → TIMETYPE

// ——————PROBLEM——————
source : PROBLEM → NODE
dest : PROBLEM → NODE
time : PROBLEM → TIME
closeness : PROBLEM → INTEGER

// ——————FEEDBACK——————
pathTaken : FEEDBACK → PATH

// ——————CASE-Index——————
source : CASE-Index → NODE
dest : CASE-Index → NODE
timeType : CASE-Index → TIMETYPE
date : CASE-Index → DATE

// ——————CASE-Content——————
path : CASE-Content → PATH

// ——————CASE-Outcome——————
frequency : CASE-Outcome → VALUE
reinforcement : CASE-Outcome → REINFORCEMENT
tripImportance : CASE-Outcome → TRIP-IMPORTANCE

// ——————CBR——————
// –Used by RANK–
weightCost : CBR → WEIGHT
weightOutcome : CBR → WEIGHT
weightTime : CBR → WEIGHT
subrankedCases : CBR → CASE − *Set*

// –Used by EXTRACT–
newPathSet : POST_SOL_MODULE → PATH − *Set*

// ——————————Misc——————————

$rightTime$: CASE → VALUE
$rightTime \equiv$
$\begin{cases} 1 & : timeType(time(problem)) = timeType(caseIndex(c)) \\ 0 & : otherwise \end{cases}$

$outcomeValue$: CASE → VALUE
$outcomeValue(c) \equiv \sum_{e \in \mathsf{Edges}(caseContent(c))} \sum_{f \in AC\_EDGE\_ATTR} factorValue(e, f)$

$costValue$: CASE → VALUE
$costValue(c) \equiv \sum_{e \in \mathsf{Edges}(caseContent(c))} \sum_{f \in PER\_EDGE\_ATTR} factorValue(e, f)$

$factorValue$: EDGE × FACTOR → VALUE
$factorValue(e, f) \equiv$
$\begin{cases} localFactorValue(owner, f, dest(problem), e) & : e \in outIncidentEdges(source(problem)) \\ globalFactorValue(owner, f, e) & : otherwise \end{cases}$

// Local Factor Value
$localFactorValue$: SEM × FACTOR × NODE × EDGE → FACTOR_VALUE
$localFactorValue(a, f, dest, e) \equiv$
$\begin{cases} angle(dest, e) * factorWeight(a, f) & : f = angle \\ 0 & : f = numberOfStops \\ interpret(geoEdgeAttr(e, f)) * factorWeight(a, f) & : f \in \text{GEO\_EDGE\_ATTRIBUTE} \\ & \quad \wedge currentNode(a) = head(e) \\ subjEdgeAttr(parentAgent(a), e, f) * factorWeight(a, f) & : otherwise. \end{cases}$

// Global Factor Value
$globalFactorValue$: SEM × FACTOR × EDGE → FACTOR_VALUE
$globalFactorValue(a, f, e) \equiv$
$\begin{cases} 0 & : f = angle \\ 1 * factorWeight(a, f) & : f = numberOfStops \\ subjEdgeAttr(parentAgent(a), e, f) * factorWeight(a, f) & : otherwise. \end{cases}$

Rules

// ——————————— IDENTIFY ———————————
IDENTIFY($self$: CBR, $problem$: PROBLEM) ≡
$source(identifiedIndex) := source(problem)$
$dest(identifiedIndex) := dest(problem)$
$timeType(identifiedIndex) := timeType(time(problem))$
$date(identifiedIndex) := date(time(problem))$

where
$identifiedIndex \equiv identifiedIndex(self)$

```
// ———————— MATCH ————————
MATCH(self : CBR, identifiedIndex : CASE-Index) ≡
  forall c ∈ caseBase(self) with exactMatch(c, identifiedIndex)
      add c to matchedCases(self)
  seq
  if matchedCases(self) = {} then
    let hops = 1 in
     while matchedCases(self) = {} ∧ hops < closeness(problem(self))
       forall c ∈ caseBase(self) with partialMatch(c, identifiedIndex, hops)
         add c to matchedCases(self)
       hops = hops + 1

where
  exactMatch(c, identifiedIndex) ≡ source(caseIndex(c)) = source(identifiedIndex)
                                 ∧ dest(caseIndex(c)) = dest(identifiedIndex)
  partialMatch(c, identifiedIndex, hops) ≡ source(caseIndex(c)) = source(identifiedIndex)
            ∧ Size(bestPath(dest(caseIndex(c)), dest(identifiedIndex))) <= hops


// ———————— RANK ————————
RANK(self : CBR, matchedCases : CASE − Set) ≡
  DoSubRank()
  seq
  choose c in subrankedCases(self) with highestDate(c)
     ballparkSolution(self) := path(caseContent(c))

where
  highestDate(c) ≡ ∀x(x ∈ subrankedCases) ⇒ date(caseIndex(c)) ≥ date(caseIndex(x))


DoSubRank(self : CBR) ≡
  forall c in positiveCases with highestSubRank(c)
    add c to subrankedCases(self)

where
  highestSubRank(c) ≡ ∀x(x ∈ positiveCases) ⇒ subRank(c) ≥ subRank(x)
  positiveCases ≡ {all n|n ∈ matchedCases(self)
                  ∧ reinforcement(caseOutcome(c)) ≠ negative}
  subRank(z) ≡ weightCost(self) * CostValue(z)+
               weightOutcome(self) * outcomeValue(z)+
               weightTime(self) * rightTime(z)


// ———————— isCopy ————————
isCopy(self : CBR) ≡
   return true
```

```
// Instantiating POST_SOL_MODULE into SEM_POST_SOL_MODULE
domain SEM_POST_SOL_MODULE ≡ POST_SOL_MODULE  where

// ———————— integrateFeedback ————————
integrateFeedback(fa : FEEDBACK_AVAILABLE)
  if takenPath(externalFeedback(fa)) <> [] then
      return true
  else
      return false

// ———————— addAsUnevalCase ————————
addAsUnevalCase(self : POST_SOL_MODULE)
  choose x ∈ unevalCaseSet(self) with unevalProblem(x) = problem(parentCBR(self))
                    ∧ unevalSolution(x) = finalSolution(parentCBR(self))
      return false
  if none
      return true

// ———————— ANALYZE ————————
ANALYZE(self : POST_SOL_MODULE, unevalCase : UNEVAL_CASE) ≡
  repairNeeded(self) := false

// ———————— EXTRACT ————————
EXTRACT(self : POST_SOL_MODULE, unevalCase : UNEVAL_CASE) ≡
                    // set of all possible paths(Node-Seq)
  let newPathSet(self) ← GetAllPermutations(pathTaken(feedback(unevalCase))) in
  forall path in newPathSet(self)
    extend CASE with extractedCase
      caseIndex(extractedCase) := Extract_Index(path)
      caseContent(extractedCase) := Extract_Content(path)
      caseOutcome(extractedCase) := Extract_Outcome(path)
      add extractedCase to extractedInfo(unevalcase(self))

Extract_Content(path : NODE − Seq) ≡
  extend CASE-Content with extractedContent
    path(extractedContent) := path
    return extractedContent
```

Extract_Index($path$: NODE − Seq) ≡
  extend CASE-Index **with** $extractedIndex$
    $source(extractedIndex) := source(path)$
    $dest(extractedIndex) := dest(path)$
    $timeType(extractedIndex) := timeType(time(unevalProblem(unevalCase)))$
    $date(extractedIndex) := date(time(unevalProblem(unevalCase)))$
    return $extractedIndex$

Extract_Outcome($path$: NODE − Seq) ≡
  extend CASE-Outcome **with** $extractedOutcome$
    $frequency(extractedOutcome) :=$ GETFREQUENCY()
    $reinforcement(extractedOutcome) :=$ GETREINFORCEMENT()
    $tripImportance(extractedOutcome) :=$ GETTRIPIMPORTANCE()
    return $extractedOutcome$

// ——— INTEGRATE ———
INTEGRATE($self$: POST_SOL_MODULE, $extractedInfo$: CASE − Set) ≡
 forall $newCase$ **in** $extractedInfo(unevalCase(self))$
   choose $oldCase$ **from** $caseBase$ **with** $matchExists(oldCase, newCase)$
     UpdateCase($oldCase, newCase$)
   if none
     add $newCase$ **to** $caseBase$

  where
  $matchExists(oldCase, newCase) \equiv$
    $source(caseIndex(oldCase)) = source(caseIndex(newCase)) \wedge$
    $dest(caseIndex(oldCase)) = dest(caseIndex(newCase)) \wedge$
    $timeType(caseIndex(oldCase)) = timeType(caseIndex(newCase)) \wedge$
    $path(caseContent(oldCase)) = path(caseContent(newCase))$
  $caseBase \equiv caseBase(parentCBR(self))$

Update_Case($oldCase$: CASE, $newCase$: CASE) ≡
  $date(caseIndex(oldCase)) := date(caseIndex(newCase))$
  $frequency(caseOutcome(oldCase)) := frequency(caseOutcome(newCase)) + 1$
  $reinforcement(caseOutcome(oldCase)) :=$ AVG($oldCase, newCase$)
  $tripImportance(caseOutcome(oldCase)) :=$ AVG($oldCase, newCase$)

B.2 Path Explorer Submachine

B.2.1 Level 0

Definitions

// ——————— Kept in 'Volatile Memory' ———————
edgePref : SEM × EDGE × NODE → PREF_VALUE
readyToExplore : SEM → BOOLEAN
bestPath : SEM × NODE × NODE → PATH

Rules

// ——————— Path Explorer SubMachine ———————
GET_SUGGESTED_PATH$_{Explorer}$(*sem* : SEM, *currentNode* : NODE, *destNode* : NODE) ≡
  if *readyToExplore* **then**
    choose *edge* **in** *outIncidentEdges*(*currentNode*) **with** *maxPref*(*edge*)
      return *concat*(*currentNode*, *bestPath*(*tail*(*edge*), *destNode*))
  else
    GLOBAL_RE_CALC
    readyToExplore := *true*
where
  maxPref(*edge*) ≡ ∀*e*(*e* ∈ *outIncidentEdges*(*currentNode*)
                ⇒ *edgePref*(*sem*, *edge*, *destNode*) ≥ *edgePref*(*sem*, *e*, *destNode*))
initialize : *readyToExplore* = *false*

B.2.2 Level 1

Definitions

// ——————— Kept in 'Profile' ———————
globalWeight : SEM → WEIGHT_VALUE
localWeight : SEM → WEIGHT_VALUE

// ——————— Kept in 'Volatile Memory' ———————
localEdgePref : SEM × EDGE × NODE → PREF_VALUE
globalPathPref : SEM × NODE × NODE → PREF_VALUE

// —edgePref—
edgePref(*a*, *e*, *dest*) ≡
  globalWeight(*a*) ∗ *globalPathPref*(*a*, *tail*(*e*), *dest*)+
  localWeight(*a*) ∗ *localEdgePref*(*a*, *e*, *dest*)

B.2.3 Level 2

Definitions

// ——————— Domains ———————
domain INDUCED_FACTOR
domain FACTOR ≡ SUBJ_EDGE_ATTR ∪ INDUCED_FACTOR

// Induced Factors
numberOfStops : → INDUCED_FACTOR
angle : → INDUCED_FACTOR

// ——————— Kept in 'Volatile Memory' ———————
// Factor Weights
factorWeight : SEM × FACTOR → FACTOR_WEIGHT

// Local Edge Pref
$localEdgePref(a, e, dest) \equiv \sum_{f \in FACTOR} localFactorValue(a, f, dest, e)$
// Local Factor Value
localFactorValue : SEM × FACTOR × NODE × EDGE → FACTOR_VALUE
$localFactorValue(a, f, dest, e) \equiv$

$$\begin{cases} angle(dest, e) * factorWeight(a, f) & : f = angle \\ 1 & : f = numberOfStops \\ interpret(geoEdgeAttr(e, f)) * factorWeight(a, f) & : f \in \text{GEO\_EDGE\_ATTR} \wedge \\ & \quad [\, currentNode(a) = head(e) \\ & \quad \vee currentNode(a) = tail(e))] \\ subjEdgeAttr(parentAgent(a), e, f) * factorWeight(a, f) & : otherwise. \end{cases}$$

// Global Edge Pref
$gEdgePref(a, e) \equiv \sum_{f \in FACTOR} globalFactorValue(a, f, e)$
// Global Factor Value
globalFactorValue : SEM × FACTOR × EDGE → FACTOR_VALUE
$globalFactorValue(a, f, e) \equiv$

$$\begin{cases} 0 & : f = angle \\ 1 * factorWeight(a, f) & : f = numberOfStops \\ interpret(geoStaticEdgeAttr(e, f)) * factorWeight(a, f) & : f \in \text{GEO\_STAT\_EDGE\_ATTR}. \\ subjEdgeAttr(parentAgent(a), e, f) * factorWeight(a, f) & : otherwise. \end{cases}$$

Rules

// ——————Global Re-Calculation——————
GLOBAL_RE_CALCULATION ≡
  forall *s* **in** *nodeSet*(*env*)
    Calculate_Best_Path(*self*, *s*)

```
// ---------- Dijkstra's ----------
Calculate_Best_Path(sem : SEM, origin : NODE, type : PREF_TYPE) ≡
  local pref [forall a in nodeSet
                pref(a) := 0 ]
  local tempNodeSet := ∅
  (forall n in nodeSet
     forall m in nodeSet
       bestPath(sem, n, m, type) := ∅)
  seq
  ( while (tempNodeSet ≠ nodeSet)// Using TurboASM constructs
     {let u = NodeWithMinLabel(nodeSet − tempNodeSet)
       add u to tempNodeSet
       bestPathPref(sem, origin, u, type) := pref(u)
       bestPath(sem, origin, u, type) := concat(bestPath(sem, origin, u, type), u)
       forall v in (nodeSet − tempNodeSet) with adjacent(u, v)
         Update_Pref(sem, u, v, type)})
where
  nodeSet ≡ nodeSet(env)
  nodeWithMinLabel(aNodeSet) ≡ p where p ∈ aNodeSet ∧
                    ∀s(s ∈ (aNodeSet − p)) ⇒ pref(p) ≥ pref(s)

// ---------- Update_Cost ----------
Update_Pref(sem : SEM, u : NODE, v : NODE, type : PREF_TYPE) ≡
  let u_v = gEdgePref(sem, edge(u, v), type)
    if pref(u) + u_v > pref(v) then
      pref(v) := pref(u) + u_v
where
  edge(u, v) ≡ a where a ∈ edgeSet(env) ∧ edgeHead(a) = u ∧ edgeTail(a) = v
```

Appendix C

Executable AsmL Model

The Executable AsmL Model is contained in the attached CD with file name "ASMLModel.doc". It is composed of approximately 3,000 lines of code (LOC) and is written using the MS Word editor.
It can be viewed using the MS Word editor or MS Visual Studio. To run the code, you need to install .Net Framework and AsmL on your machine. See also the attached README file for help.
As described in Chapter 10, the AsmL model is organized as five distinct parts.

C.1 Global Definitions

Global Definitions are contained in the file "ASMLModel.doc" as Section 1. These are as follows:
Section 1.1: LINKING SOCIAL SYSTEMS TO DASM MODELS
Section 1.2: ENVIRONMENT REPRESENTATION
Section 1.3: PERSON AGENT
Section 1.4: SIGNALS
Section 1.5: OTHER GLOBAL FUNCTIONS (TIME, etc)

C.2 AsmL Abstract Model

AsmL Abstract Model is contained in the file "ASMLModel.doc" as Section 2. It is comprised of the following:
Section 2.1: SPACE EVOLUTION MODULE (SEM)
Section 2.2: TARGET SELECTION MODULE (TSM)
Section 2.3: AGENT DECISION MODULE (ADM)

C.3 AsmL Refined Model

AsmL Refined Model is contained in the file "ASMLModel.doc" as Section 3. It is comprised of the following:
Section 3.1: CASE BASED RESONER (CBR) - Abstract and Concrete
Section 3.2: PATH EXPLORER SUBMACHINE

C.4 Execution Specific Additions

Execution Specific Additions are contained in the file "ASMLModel.doc" as Section 4.

C.5 Visualization Specific Additions

Visualization Specific Additions are contained in the file "ASMLModel.doc" as Section 5. These are comprised of the following:
Section 5.1: DATA STRUCTURE OF XML FILE.
Section 5.2: COMMUNICATION COMMANDS (between the AsmL Model and Visualization)

Bibliography

[1] Agnar Aamodt and Enric Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Commun.*, 7(1):39–59, 1994.

[2] M. A. Anwar and T. Yoshida. Integrating OO Road Network Database, Cases and Knowledge for Route Finding. In *SAC '01: Proceedings of the 2001 ACM Symposium on Applied Computing*, pages 215–219, New York, NY, USA, 2001. ACM Press.

[3] S. Bankes. Agent-based Modeling: A Revolution? *Proceedings of the National Academy of Science of the United States of America*, 99:7199–7200, 2002.

[4] Daniel M. Berry. Formal Methods: The Very Idea Some Thoughts About Why They Work When They Work. *Electronic Notes in Theoretical Computer Science*, 25, 1999.

[5] E. Börger. The ASM Ground Model Method as a Foundation for Requirements Engineering. *Verification: Theory and Practice*, pages 145–160, 2003.

[6] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.

[7] Egon Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15(2-3):237–257, 2003.

[8] A. E. Bottoms and P. Wale. Environmental Criminology. In *The Oxford Handbook of Criminology (3rd ed.)*, pages 620—656. Oxford University Press, 2002.

[9] P. J. Brantingham and P. L. Brantingham. A Theoretical Model of Crime Site Selection. In M. D. Krohn and R. L. Akers, editors, *Crime, Law and Sanctions.* Sage Publications Inc., 1978.

[10] P. J. Brantingham and P. L. Brantingham. *Patterns in Crime.* New Tork: Macmillan Publishing Company, 1984.

[11] P. J. Brantingham and P. L. Brantingham. Introduction: The Dimensions of Crime. In P. J. Brantingham and P. L. Brantingham, editors, *Environmental Criminology*, pages 7–26. Waveland Press, 1991.

[12] P. J. Brantingham and P. L. Brantingham. Introduction to the 1991 Reissue: Notes on Environmental Criminology. In P. J. Brantingham and P. L. Brantingham, editors, *Environmental Criminology*, pages 7–26. Waveland Press, 1991.

[13] P. J. Brantingham and P. L. Brantingham. Notes on the Geometry of Crime. In P. J. Brantingham and P. L. Brantingham, editors, *Environmental Criminology*, pages 7–26. Waveland Press, 1991.

[14] P. J. Brantingham and P. L. Brantingham. Environment, Routine and Situation: Toward a Pattern Theory of Crime. *Advances in Criminological Theory*, pages 259–294, 1993.

[15] P. J. Brantingham and P. L. Brantingham. Nodes, Paths and Edges: Considerations on the Complexity of Crime and the Physical Environment. *Journal of Environmental Psychology*, pages 3–28, 1993.

[16] P. J. Brantingham and P. L. Brantingham. Computer Simulation as a Tool for Environmental Criminologists. *Security Journal*, pages 22–30, 2004.

[17] P. J. Brantingham, P. L. Brantingham, and U. Glässer. Computer Simulation in Criminal Justice Research. *Criminal Justice Matters*, (58), February 2005.

[18] P. L. Brantingham, U. Glässer, B. Kinney, K. Singh, and M. Vajihollahi. A Computational Model for Simulating Spatial Aspects Crime in Urban Environments. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Oct 2005.

[19] P. L. Brantingham, U. Glässer, B. Kinney, K. Singh, and M. Vajihollahi. Mastermind: Modeling and Simulation of Criminal Activity in Urban Environments. Technical Report SFU-CMPT-TR-2005-01, Simon Fraser University, Feb 2005.

[20] P. L. Brantingham, U. Glässer, B. Kinney, K. Singh, and M. Vajihollahi. Modeling Urban Crime Patterns: Viewing Multi-Agent Systems as Abstract State Machines. In E. Brger D. Beauquier and A. Slissenko, editors, *Proc. 12th International Workshop on Abstract State Machines*, pages 101–117, March 2005.

[21] P. L. Brantingham, U. Glässer, K. Singh, and M. Vajihollahi. Mastermind: Modeling and Simulation of Criminal Activity in Urban Environments. Technical Report SFU-CMPT-TR-2005-14, Simon Fraser University, July 2005. Revised version of SFU-CMPT-TR-2005-01, February 2005.

[22] M. E. Bratman, D. Israel, and M. Pollack. Plans and Resource-Bounded Practical Reasoning. In R. Cummins and J. L. Pollock, editors, *Philosophy and AI: Essays at the Interface*, pages 1–22. The MIT Press, Cambridge, Massachusetts, 1991.

[23] M. E. Bratman, D. Israel, and M. E. Pollack. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4:349–355, 1988.

[24] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.

[25] D. Canter. *Criminal Shadows.* Authorlink, 1994.

[26] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Program*, 73(2):129–174, 1996.

[27] R. V. Clarke. Situational Crime Prevention: Its Theoretical Basis and Practical Scope. In M. Tonry and N. Morris, editors, *Crime and Justice: An Annual Review of Research*, pages 225–256, 1983.

[28] L. E. Cohen and M. Felson. Social Change and Crime Rate Trends: A Routine Activity Approach. *American Sociological Review*, pages 588–608, 1979.

[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction To Algorithms, Second Edition.* MIT Press, September 2001.

[30] D. Cornish and R. V. Clarke. Introduction. In D. Cornish and R. V. Clarke, editors, *The Reasoning Criminal: Rational Choice Perspectives on Offending, Research in Criminology*, pages 1–16. New York: Springer-Verlag, 1986.

[31] P. Davidsson. Agent Based Social Simulation: A Computer Science View. *Journal of Artificial Societies and Social Simulation*, 5, January 2002.

[32] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* Formal Approaches to Computing and Information Technology. Springer, May 2001.

[33] E. W. Dijkstra. A Note On Two Problems In Connection With Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[34] M. d'Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan, and M. Wooldridge. Formalisms For Multi-Agent Systems. In *In First UK Workshop on Foundations of Multi-Agent Systems*, 1996.

[35] M. d'Inverno and M. Luck. Formal Agent Development: Framework to System. In J. L. Rash, C. Rouff, W. Truszkowski, D. F. Gordon, and M. G. Hinchey, editors, *Formal Approaches to Agent-Based Systems, First International Workshop, FAABS 2000. Revised Papers*, volume 1871 of *Lecture Notes in Computer Science.* Springer, 2001.

[36] J. Doran, M.Palmer, N. Gilbert, and P. Mellars. The EOS Project: Modelling Upper Palaeolithic Social Change. In N. Gilbert and J. Doran, editors, *Simulating Society: The Computer Simulation of Social Phenomena*, chapter 9, pages 195–221. UCL Press, London, 1994.

[37] A. Drogoul and J. Ferber. Multi-Agent Simulation as a Tool for Studying Emergent Processes in Societies. In N. Gilbert and J. Doran, editors, *Simulating Society: The Computer Simulation of Social Phenomena*, chapter 6, pages 127–142. UCL Press, London, 1994.

[38] A. Drogoul, D. Vanbergue, and T. Meurisse. Multi-Agent Based Simulation: Where Are the Agents? *Journal of Artificial Societies and Social Simulation*, 6, July 2002.

[39] J. E. Eck and D. Weisburd. Crime Places in Crime Theory. In *Crime and Place, Crime Prevention Studies*, pages 1–33. The Police Executive Research Forum, 1995.

[40] P. Ekblom and N. Tilley. Going Equipped: Criminology, Situational Crime Prevention and the Resourceful Offender. *British Journal of Criminology*, pages 376–398, 2000.

[41] Roozbeh Farahbod. Extending and Refining an Abstract Operational Semantics of the Web Services Architecture for the Business Process Execution Language. Master's thesis, Simon Fraser University, Burnaby, Canada, July 2004.

[42] M. Felson. Routine Activities and Crime Prevention in the Developing Metropolis. *Criminology*, pages 911–931, 1987.

[43] N.E. Fenton and S.L. Pfleeger. *Software Metrics: Rigorous and Practical Approach.* International Thomson Press, 1997.

[44] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence.* Addison Wesley, February 1999.

[45] M. Fisher. If Z is he Answer, What Could the Question Possibly Be? In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, volume 1193, pages 65–66. Springer-Verlag: Heidelberg, Germany, 12–13 1997.

[46] M. Fisher and M. Wooldridge. Towards Formal Methods for Agent-Based Systems. In D. Duke and A. Evans, editors, *BCS-FACS Northern Formal Methods Workshop. Electronic Workshops in Computing.* Springer Verlag, 1997.

[47] Martin Fowler. The New Methodology, April 2003.

[48] S. Franklin and A. Gasser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages.* Springer-Verlag, 1996.

[49] G. Gallo and S. Pallottino. Shortest Paths Algorithms. *Annals of Operations Research*, 13:3–79, 1988.

[50] M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. The Belief-Desire-Intention Model of Agency. In J. Müller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 1–10. Springer-Verlag: Heidelberg, Germany, 1999.

[51] N. Gilbert and S. Bankes. Platforms and Methods for Agent-Based Modeling. *Proceedings of the National Academy of Science of the United States of America*, 99:7197–7198, 2002.

[52] N. Gilbert and J. Doran. *Simulating Societies.* UCL Press, 1995.

[53] N. Gilbert and K. G. Troitzsch. *Simulation for the Social Scientist.* Open University Press, 1999.

[54] U. Glässer, Y. Gurevich, and M. Veanes. An Abstract Communication Model for Distributed Systems. *IEEE Transactions on Software Engineering*, 30(7):458–472, 2004.

[55] A. R. Golding and P. S. Rosenbloom. Improving Rule-Based Systems Through Case-Based Reasoning. In *National Conference on Artificial Intelligence*, pages 22–27, 1991.

[56] L. Gunderson and D.Brown. Using a Multi-Agent Model to Predict Both Physical and Cyber Criminal Activity. *IEEE International Conference on Systems, Man, and Cybernetics*, 4:2338–2343, 2000.

[57] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[58] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[59] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[60] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[61] V. Hilaire, A. Koukam, P. Gruer, and J. P. Müller. Formal Specification and Prototyping of Multi-agent Systems. *Lecture Notes in Computer Science*, 1972, 2001.

[62] Jan Husdal. Fastest Path Problems in Dynamic Transportation Networks, 2000.

[63] N. Jing., Y. W. Huang, and E. A. Rundensteiner. Hierarchical Optimization of Optimal Path Finding For Transportation Applications. In *Proceedings of the Fifth International Conference on Information and Knowledge Management*, pages 261–268. ACM Press, 1996.

[64] K. Koffka. *Principles of Gestalt Psychology.* Harcourt, 1967.

[65] J. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann Publishers Inc., 1993.

[66] J. Kolodner and D. Leake. A Tutorial Introduction to Case-Based Reasoning. In D. Leake, editor, *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, pages 31–65. AAAI Press, 1996.

[67] David B. Leake. *Case-Based Reasoning: Experiences, Lessons and Future Directions.* MIT Press, 1996.

[68] B. Liu. Intelligent Route Finding: Combining Knowledge and Cases and an Efficient Search Algorithm. In *European Conference on Artificial Intelligence*, pages 380–384, 1996.

[69] B. Liu, S.H. Choo, S.L. Lok, S.M. Leong, S.C. Lee, F.P. Poon, and H.H. Tan. Integrating Knowledge-Based Approach, Case-Based Reasoning and Dijkstra Algorithm for Routing Finding. In *Proceedings of The Tenth IEEE Conference on Artificial Intelligence for Applications*, pages 149–155, 1994.

[70] B. Liu and J. Tay. Using Knowledge about the Road Network for Route Finding. In *In IEEE Transactions on Systems, Man and Cybernetics*, volume 27. IEEE, July 1997.

[71] M. Luck and M. d'Inverno. A Formal Framework for Agency and Autonomy. In V. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 254–260, San Francisco, CA, USA, 1995. AAAI Press.

[72] M. L. Maher, M. Balachandran, and D. M. Zhang. *Case-Based Reasoning in Design.* Lawrence Erlbaum Associates, 1995.

[73] C. Marling, M. Sqalli, E. Rissland, H. Munoz-Avila, and D. Aha. Case-Based Reasoning Integrations. *AI Mag.*, 23(1):69–86, 2002.

[74] Microsoft Foundations of Software Engineering Group. *The Abstract State Machine Language.* Last visited June 2005, http://research.microsoft.com/fse/asml/.

[75] C. E. Noon and F. B. Zhan. Shortest Path Algorithms: An Evaluation Using Real Road Networks. *Transportation Science*, 1996.

[76] Bashar Nuseibeh and Steve Easterbrook. Requirements Engineering: A Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[77] K. Ramamohanarao P. Busetta, J. Bailey. A Reliable Computational Model for BDI Agents. *Proceedings of 1st International Workshop on Safe Agents*, July 2003.

[78] A. J. Patel. Amit's Thoughts on Path Finding and A-Star, 2003. http://theory.stanford.edu/~amitp/GameProgramming/.

[79] J. Prentzas and I. Hatzilygeroudis. Integrations of Rule-Based and Case-Based Reasoning. In *In Proceedings of the International Conference on Computer, Communication and Control Technologies (CCCT-03)*, volume 4, pages 81–85, 2003.

[80] G. F. Rengert. Burglary in Philadelphia: A Critique of an Opportunity Structure Model. In *Environmental Criminology*, pages 189–201. Waveland Press Inc., 1991.

[81] G. F. Rengert. The Journey to Crime: Conceptual Foundations and Policy Implications. In *Crime, Policing and Place: Essays in Environmental Criminology*, pages 109–117. Routlede, 1992.

[82] Christopher K. Riesbeck and Roger C. Schank. *Inside Case-Based Reasoning.* Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1989.

[83] D. K. Rossmo. *Geographic Profiling.* CRC Press, 2000.

[84] A. Seror. Simulation of Complex Organizational Processes: A Review of Methods and Their Epistemological Foundations. In N. Gilbert and J. Doran, editors, *Simulating Societies.* UCL Press, 1994.

[85] O. Shehory and A. Sturm. Evaluation of Modeling Techniques for Agent-Based Systems. In *Proceedings of the Fifth International Conference on Autonomous agents*, pages 624–631. ACM Press, 2001.

[86] J. Sonnenfeld. Geography, Perception and the Behavioral Environment. In P. W. English and R. C. Mayfield, editors, *Man, Space and the Environment*, pages 244–251. Oxford University Press, New York, 1972.

[87] P. J. Starr. Modeling Issues and Decisions in System Dynamics. In JR A. A. Legasto, J. W. Forrester, and J. M. Lyneis, editors, *System Dynamics*. North Holland, 1980.

[88] J. Sungwon and S. Pramanik. An Efficient Path Computation Model For Hierarchically Structured Topographical Road Maps. *IEEE Transactions on Knowledge and Data Engineering*, 14:1029–1046, Sep/Oct 2002.

[89] K. P. Sycara. Multiagent Systems. *AI Magazine*, pages 79–92, 1998.

[90] G. Wagner. Practical Theory and Theory-Based Practice. In J. P. Müller, M. J. Wooldridge, and N. R. Jennings, editors, *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, volume 1193, pages 67–69. Springer-Verlag: Heidelberg, Germany, 12–13 1997.

[91] M. Wooldridge. Intelligent Agents: The Key Concepts. In M. Luck, V. Marik, O. Stepankova, and R. Trappl, editors, *Multi-Agent Systems and Applications II*, pages 3–43. Springer, 2001.

[92] M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley and Sons Ltd, 2002.

[93] R. T. Wright and S. H. Decker. *Armed Robbers in Action: Stickups and Street Culture*. Northeastern University Press, 1997.

[94] W. Yimin, X. Jianmin, H. Yucong, and Y. Qinghong. A Shortest Path Algorithm Based on Hierarchical Graph Model. *Intelligent Transportation Systems*, 2:1511–1514, October 2003.

[95] B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2000.

[96] F. B. Zhan. Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures. *Journal of Geographic Information and Decision Analysis*, 1(1):69–82, February 1997.